CrossTalk

June 2008 **The Journal of Defense Software Engineering** Vol. 21 No. 6

Software Quality

DON'T SKIP PAGE 17!

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **JUN 2008**	2. REPORT TYPE	3. DATES COVERED **00-00-2008 to 00-00-2008**

4. TITLE AND SUBTITLE **CrossTalk: The Journal of Defense Software Engineering. Volume 21, Number 6, June 2008**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **OO-ALC/MASE,6022 Fir Ave,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**32**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

On The Cover

Cover Design by
Kent Bingham

Additional art services provided by Janna Jensen

CrossTalk

Co-Sponsors:	
DoD-CIO	*The Honorable John Grimes*
OSD (AT&L)	*Kristen Baldwin*
NAVAIR	*Jeff Schwalb*
76 SMXG	*Phil Perkins*
309 SMXG	*Karl Rogers*
DHS	*Joe Jarzombek*
Staff:	
Managing Director	*Brent Baxter*
Publisher	*Kasey Thompson*
Managing Editor	*Ken Davies*
Associate Editor	*Chelene Fortier-Lozancich*
Article Coordinator	*Nicole Kentta*
Phone	(801) 775-5555
E-mail	stsc.customerservice@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/crosstalk

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the Department of Defense Chief Information Office (DoD-CIO); the Office of the Secretary of Defense (OSD) Acquisition, Technology and Logistics (AT&L); U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). DoD-CIO co-sponsor: Assistant Secretary of Defense (Networks and Information Integration). OSD (AT&L) co-sponsor: Software Engineering and System Assurance. USN co-sponsor: Naval Air Systems Command. USAF co-sponsors: Oklahoma City-Air Logistics Center (ALC) 76 Software Maintenance Group (SMXG); and Ogden-ALC 309 SMXG. DHS co-sponsor: National Cyber Security Division of the Office of Infrastructure Protection.

The USAF Software Technology Support Center (STSC) is the publisher of CrossTalk, providing both editorial oversight and technical review of the journal. CrossTalk's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 30.

517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CrossTalk does not pay for submissions. Articles published in CrossTalk remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CrossTalk.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CrossTalk are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857 or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

Quality Programming Begets Software Quality

Joe Jarzombek, Director for Software Assurance in the National Cyber Security Division (NCSD) of the Department of Homeland Security (DHS, CrossTalk's co-sponsor), has given many keynote presentations at conferences in which he advocates the need for security-enhanced processes and practices. His message at the Software Engineering Process Group Conference in March was a snapshot of the session he facilitated on February 8, 2008, on "Security-Enhanced Quality Assurance and Project Management: Mitigating Risks to the Enterprise" at the Defense Acquisition University's (DAU) Advanced Software Acquisition Management course. His opening message set the theme for his DAU presentation and this issue of CrossTalk. Jarzombek said, "With today's global supply chain for information technology and software, the processes associated with software engineering, quality assurance (QA), and project management must explicitly address security risks posed by exploitable software. However, traditional processes do not explicitly address software security risks that can be passed from projects to using organizations.

"Mitigating supply chain risks requires an understanding and management of suppliers' process capabilities, products and services. Enterprise risks stemming from the supply chain are influenced by suppliers and acquisition projects (including procurement, QA, and testing). Software assurance processes and practices span development and acquisition.

"Derived (non-explicit) security requirements should be elicited and considered. QA and testing can integrate security considerations in their practices to enhance value in mitigating risks to the enterprise."

He then asked the audience, "What legacy do you intend to leave from the programs in which you have project responsibilities? Is it one that contributes to a more resilient system and enterprise or one that was simply good enough to get the customer to accept without an understanding of the residual risk passed to the end user?"

Along those lines, this month's issue of CrossTalk deals with software quality. In his article *The Software Quality Challenge*, Watts S. Humphrey discusses how today's more complex software offers greater challenges in the areas of safety, security and reliability. Capers Jones talks about two measures that have a strong influence on the outcomes of software projects in *Measuring Defect Potentials and Defect Removal Efficiency*. Asking the tantalizing question, "Would your company like to save $100,000 per day?" Thomas D. Neff offers insights into how that is possible in *Quality Processes Yield Quality Products*; while Dr. Paul Anderson discusses the advantages and limitations of using static-analysis tools in *The Use and Limitations of Static-Analysis Tools to Improve Software Quality*. Next, D. Richard Kuhn, Dr. Yu Lei, and Dr. Raghu Kacker offer new tools for automating the production of complete test cases covering up to 6-way combinations in *Automated Combinatorial Test Methods – Beyond Pairwise Testing*, and Dr. Jeffrey Voas gives us a look at the misunderstood term *software quality* in *Software Quality Unpeeled*.

For additional information, I want to remind everybody that DHS' NCSD offers free resources related to *security-enhanced quality assurance, project management, and software engineering* via their software assurance and BuildSecurityIn Web sites at: <www.us-cert.gov/swa> and <https://buildsecurityin.us-cert.gov>. More can also be learned at the World Congress for Software Quality, a major international gathering of software quality professionals that will take place September 15-18, 2008 in Bethesda, Maryland. For more information on the conference, visit <www.asq.org/conferences/wcsq>.

Finally, I would like to thank Beth Starrett for the exemplary work she has done over the past eight years as publisher of CrossTalk. During her tenure at the helm the publication has thrived, and has broadened both its scope and depth in the field of defense software engineering. With her departure, we welcome Kasey Thompson as our new publisher. Kasey has long been associated with CrossTalk, and we look forward to a new era of cutting-edge articles, innovative features, and ever-evolving quality information on the subjects that you, our readers, demand.

[illegible]

Brent D. Baxter
Managing Director

The Software Quality Challenge

Watts S. Humphrey
The Software Engineering Institute

Many aspects of our lives are governed by large, complex systems with increasingly complex software, and the safety, security, and reliability of these systems has become a major concern. As the software in today's systems grows larger, it has more defects, and these defects adversely affect the safety, security, and reliability of the systems. This article explains why the common test-and-fix software quality strategy is no longer adequate, and characterizes the properties of the quality strategy we must pursue to solve the software quality problem in the future.

Today, many of the systems on which our lives and livelihoods depend are run by software. Whether we fly in airplanes, file taxes, or wear pacemakers, our safety and well being depend on software. With each system enhancement, the size and complexity of these systems increase, as does the likelihood of serious problems. Defects in video games, reservations systems, or accounting programs may be inconvenient, but software defects in aircraft, automobiles, air traffic control systems, nuclear power plants, and weapons systems can be dangerous.

Everyone depends on transportation networks, hospitals, medical devices, public utilities, and the international financial infrastructure. These systems are all run by increasingly complex and potentially defective software systems. Regardless of whether these large life-critical systems are newly developed or composed from modified legacy systems, to be safe or secure, they must have quality levels of very few defects per million parts.

Modern, large-scale systems typically have enormous requirements documents, large and complex designs, and millions of lines of software code. Uncorrected errors in any aspect of the design and development process generally result in defects in the operational systems. The defect levels of such operational systems are typically measured in defects per thousand lines of code. A one million line-of-code system with the typical quality level of one defect per 1,000 lines would have 1,000 undiscovered defects, while any reasonably safe system of this scale must have only a very few defects, certainly less than 10.

The Need for Quality Software

Before condemning programmers for doing sloppy work, it is appropriate to consider the quality levels of other types of printed media. A quick scan of most books, magazines, and newspapers will reveal at least one and generally more defects per page while even poor-quality software has much less than one defect per listing page. This means that the quality level of even poor-quality software is higher than that obtained for other kinds of human written text. Programming is an exacting business, and these professionals are doing extraordinarily high quality work. The only problem is that based on historical trends, future systems will be much larger and more complex than today, meaning that just to maintain today's defect levels, we must do much higher quality work in the future.

To appreciate the challenge of achieving 10 or fewer defects per million lines of code, consider what the source listing for such a program would look like. The listing for a 1,000-line program would fill 40 text pages; a million-line program would take 40,000 pages. Clearly, finding all but 10 defects in 40,000 pages of material is humanly impossible. However, we now have complex life-critical systems of this scale and will have much larger ones in the relatively near future. So we must do something, but what? That is the question addressed in this article.

Why Defective Systems Work

To understand the software quality problem, the first question we must answer is *If today's software is so defective, why aren't there more software quality disasters?* The answer is that software is an amazing technology. Once you test it and fix all of the problems found, that software will always work under the conditions for which it was tested. It will not wear out, rust, rot, or get tired. The reason there are not more software disasters is that testers have been able to exercise these systems in just about all of the ways they are typically used. So, to solve the software quality problem, all we must do is keep testing these systems in all of the ways they will be used. So what is the problem?

The problem is complexity. The more complex these systems become, the more different ways they can be used, and the more ways users can use them, the harder it is to test all of these conditions in advance. This was the logic behind the beta-testing strategy started at IBM with the OS/360 system more than 40 years ago. Early copies of the new system releases were sent to a small number of trusted users and IBM then fixed the problems they found before releasing the public version. This strategy was so successful that it has become widely used by almost all vendors of commercial software.

Unfortunately, however, the beta-testing strategy is not suitable for life-critical systems. The V-22 Osprey helicopter, for example, uses a tilting wing and rotor system in order to fly like an airplane and land like a helicopter. In one test flight, the hydraulic system failed just as the pilot was tilting the wing to land. While the aircraft had a built-in back-up system to handle such failures, the aircraft had not been tested under those precise conditions, and the defect in the back-up system's software had not been found. The defect caused the V-22 to become unstable and crash, killing all aboard.

The problem is that as systems become more complex, the number of possible ways to use these systems grows exponentially. The testing problem is further complicated by the fact that the way such systems are configured and the environments in which they are used also affect the way the software is executed. Table 1 lists some of the variations that must be considered in testing complex systems. An examination of the number of possibilities for even relatively simple systems shows why it is impractical to test all possibilities for any complex system. So why is complex software so defective?

Some Facts

Software is and must remain a human-produced product. While tools and techniques have been devised to automate the production of code once the requirements and design are known, the requirements and design must be produced by people. Further, as systems become increasingly complex, their requirements and design grow increasingly complex. This complexity then leads to errors, and these errors result in defects in the requirements, design, and the operational code itself. Thus, even if the code could be automatically generated from the defective requirements and design, that code would reflect these requirements and design defects and, thus, still be defective.

When people design things, they make mistakes. The larger and more complex their designs, the more mistakes they are likely to make. From course data on thousands of experienced engineers learning the Personal Software Process^{SM} (PSP^{SM}), it has been found that developers typically inject about 100 defects into every 1,000 lines of the code they write [1]. The distribution for the total defects injected by 810 experienced developers at the beginning of PSP training is shown by the total bars in Figure 1. While there is considerable variation and some engineers do higher-quality work, just about everybody injects defects.

Developers use various kinds of tools to generate program code from their designs, and they typically find and fix about half of their defects during this process. This means that about 50 defects per 1,000 lines of code remain at the start of initial testing. Again, the distribution of the defects found in initial testing is also shown by the test bars in Figure 1.

Developers generally test their programs until they run without obvious failures. Then they submit these programs to systems integration and testing where they are combined with other similar programs into larger and larger sub-systems and systems for progressively larger-scale testing. The defect content of programs entering systems testing typically ranges between 10 and 20 defects per 1,000 lines.

The most disquieting fact is that testing can only find a fraction of the defects in a program. That is, the more defects a program contains at test entry, the more it is likely to have at test completion. The reason for this is the point previously made about extensive testing. Clearly, if defects are randomly sprinkled throughout a large and complex software system, some of them will be in the most rarely used parts of the system and others will be in those parts that are only exercised under failure conditions. Unfortunately, these rarely used parts are the ones most likely to be exercised when such systems are subjected to the stresses of high transaction volume, accidents, failures, or military combat.

1.	Data rates
2.	Data values
3.	Data errors
4.	Configuration variations
5.	Number, type, and timing of simultaneous processes
6.	Hardware failures
7.	Network failures
8.	Operator errors
9.	Version changes
10.	Power variations

Table 1: *Some of the Possible Testing Variations*

Switches	Paths
1	2
4	6
9	20
16	70
36	924
49	3,432
64	12,870
81	48,620
100	184,756
400	1.38E+11

Table 2: *Possible Paths Through a Network*

The Defect Removal Problem

A defect is an incorrect or faulty construction in a product. For software, defects generally result from mistakes that the designers or developers make as they produce their products. Examples are oversights, misunderstandings, and typos. Furthermore, since defects result from mistakes, they are not logical. As a consequence, there is no logical or deductive process that could possibly find all of the defects in a system. They could be anywhere, and the only way to find all of the defects with testing is to exhaustively test every path, function, or system condition.

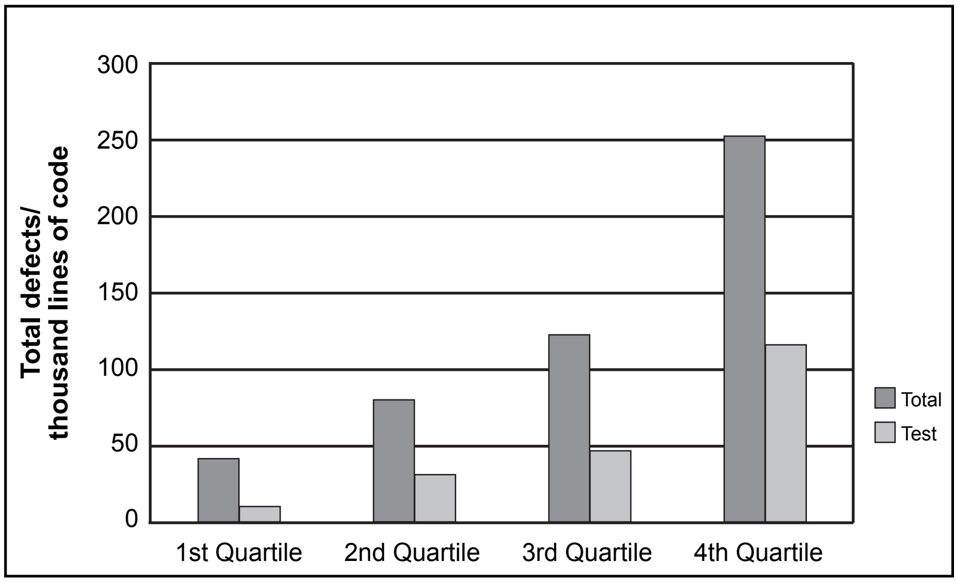
This leads to the next question which concerns the testing objective: "Must we find all of the defects, or couldn't we just find and fix those few that would be dangerous?" Obviously, we only need to fix the defects that would cause trouble, but there is no way to determine which defects these are without examining all of the defects. For example, a complex design defect that produced a confusing operator message could pose no danger while a trivial typographical mistake that changed a no to a yes could be very dangerous. Since there is no way to tell in advance which defects would be damaging, we must try to find them all. Then, after finding them, we must fix at least all of the ones that would be damaging.

The Testing Problem

Since defects could be anywhere in a large software system, the only way testing could find them all would be to completely test every segment of code in the entire program. To understand this issue, consider the program structure in Figure 2. This code fragment has one branch instruction at point B; three segments: A to B, B to C, and B to D; and two possible paths or routes through the fragment: A-B-C and A-B-D. So, for a program fragment like this, there could be defects on any of the code segments as well as in the branch instruction itself.

For a large program, the numbers of possible paths or routes through a program can vary by program type, but pro-

Figure 1: *Total and Test Defect Rates of 810 Experienced Engineers*



^{SM} Personal Software Process and PSP are service marks of Carnegie Mellon University.

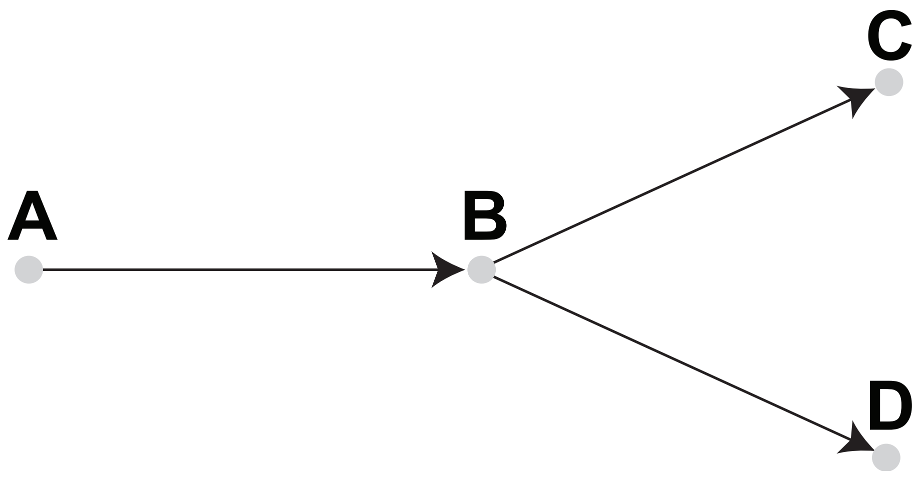


Figure 2: *A Three-Segment Code Fragment*

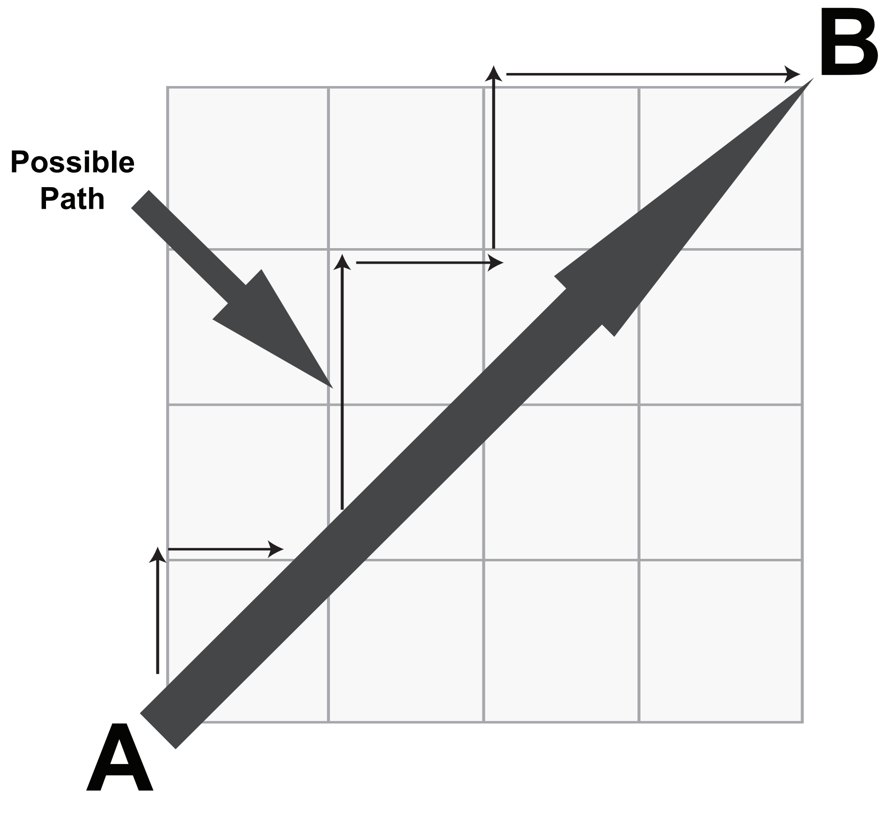
grams generally have about one branch instruction for every 10 or so lines of code. This means that a million-line program would typically have about 100,000 branch instructions. To determine the magnitude of the testing problem for such a system, we must determine the number of test paths through a network of 100,000 switches. Here, from Figure 3, we can calculate that, for a simple network of 16 switches, there are 70 possible paths from A to B. As shown in Table 2, the number of possible paths through larger networks grows rapidly with 100 switches having 184,756 possible paths and 400 switches having 1.38E+11 possible paths. Clearly, the number of possible paths through a system with 100,000 switches could, for practical purposes, be considered infinite. Furthermore, even if comprehensive path testing were possible, more than path testing would be required to uncover the defects that involved timing, synchronization, or unusual operational conditions.

Conclusions on Testing

At this point, several conclusions can be drawn. First, today's large-scale systems typically have many defects. Second, these defects generally do not cause problems as long as the systems are used in ways that have been tested. Third, because of the growing complexity of modern systems, it is impossible to test all of the ways in which such systems could be used. Fourth, when systems are stressed in unusual ways, their software is most likely to encounter undiscovered defects. Fifth, under these stressful conditions, these systems are least likely to operate correctly or reliably.

Therefore, with the current commonly used test-based software quality strategy, large-scale life-critical systems will be least reliable in emergencies – and that is when reliability is most important.

Figure 3: *Possible Paths Through a 16-Switch Network*



Successful Quality Strategies

Organizations have reached quality levels of a few defects per million parts, but these have been with manufacturing and not design or development processes. In the manufacturing context, the repetitive work is performed by machines, and the quality challenge is to consistently and properly follow all of the following steps:

- Establish quality policies, goals, and plans.
- Properly set up the machines.
- Keep the machines supplied with high-quality parts and materials.
- Maintain the entire process under continuous statistical control.
- Evaluate the machine outputs.
- Properly handle all deviations and problems.
- Suitably package, distribute, or otherwise handle the machine outputs.
- Consistently strive to improve all aspects of the production and evaluation processes.

While these eight steps suggest some approaches to consider for software development, they are not directly applicable for human-intensive work such as design and development. However, by considering an analogous approach with people instead of machines, we begin to see how to proceed.

The Eight Elements of Software Quality Management

The eight steps required to consistently produce quality software are based on the five basic principles of software quality shown in the Software Quality Principles sidebar. With these principles in mind, we can now define the eight steps required for an effective software quality initiative.

1. Establish quality policies, goals, and plans.
2. Properly train, coach, and support the developers and their teams.
3. Establish and maintain a requirements quality-management process.
4. Establish and maintain statistical control of the software engineering process.
5. Review, inspect, and evaluate all product artifacts.
6. Evaluate all defects for correction and

to identify, fix, and prevent other similar problems.

7. Establish and maintain a configuration management and change control system.
8. Continually improve the development process.

The following sections discuss each of these eight steps and relate them to the software quality principles as shown in the sidebar.

Software Quality Principles*

1. Properly managed quality programs reduce total program cost, increase business value and quality of delivered products, and shorten development times.
   - 1.1. If cost or development times increase, the quality program is not being properly implemented.
   - 1.2. The size of a product, including periodic reevaluation of size as changes occur, must be estimated and tracked.
   - 1.3. Schedules, budgets, and quality commitments must be mutually consistent and based on sound historical data and estimating methods.
   - 1.4. The development approach must be consistent with the rate of change in requirements.
2. To get quality work, the customer must demand it.
   - 2.1. Attributes that define quality for a software product must be stated in measurable terms and formally agreed to between developers and customers as part of the contract. Any instance of deviation from a specified attribute is a *defect*.
   - 2.2. The contract shall specify the agreed upon quality level, stated in terms of the acceptable quantity or ratio of deviations (defects) in the delivered product.
3. The developers must feel personally responsible for the quality of the products they produce.
   - 3.1. The development teams must plan their own work and negotiate their commitments with management and the customer.
   - 3.2. Software managers must provide appropriate training for developers.
   - 3.3. A developer is anyone who produces a part of the product, be it a designer, documenter, coder, or systems designer.
4. For the proper management of software development projects, the development teams themselves must plan, measure, and control the work.
   - 4.1. Project teams must have knowledge and experience in the relevant technologies and applications domains commensurate with project size and other risk factors.
   - 4.2. Removal yield at each step and in total pre-delivery must be measured.
   - 4.3. Effort associated with each activity must be recorded.
   - 4.4. Defects discovered by each appraisal method must be recorded.
   - 4.5. Measurements must be recorded by those performing the activity and be analyzed by both developers and managers.
5. Software management must recognize and reward quality work.
   - 5.1. Projects must utilize a combination of appraisal methods sufficient to verify the agreed defect levels.
   - 5.2. Managers must use measures to ensure high quality and improve processes.
   - 5.3. Managers must use measurements with due respect for individuals.

* These principles were defined by a group of 13 software quality experts convened by Taz Daughtrey. The experts are: Carol Dekkers, Gary Gack, Tom Gilb, Watts Humphrey, Joe Jarzombek, Capers Jones, Stephen Kan, Herb Krasner, Gary McGraw, Patricia McQuaid, Mark Paulk, Colin Tully, and Jerry Weinberg.

Step 1: Quality Policies, Goals, and Plans

Policies, goals, and plans go together and form the essential foundation for all effective quality programs. The fundamental policy that forms the foundation for the quality program is that quality is and must be *the* first priority. Many software developers, managers, and customers would argue that product function is critical and that project schedule and program cost are every bit as important as quality. In fact, they will argue that cost, schedule, and quality must be traded off.

The reason this is a policy issue is given in the first principle of software quality stated in the sidebar: *Properly managed quality programs reduce total program cost, increase business value and quality of delivered products, and shorten development times.* Customers must demand quality work from their suppliers and management must believe that if the quality program increases program costs or schedules, that quality program is not properly managed. There is, in fact, no cost/schedule/quality trade-off: manage quality properly, and cost and schedule improvements will follow. Everyone in the organization must understand and accept this point: it is always faster and cheaper to do the job right the first time than it is to waste time fixing defective products after they have been developed.

Once the basic quality policy is in place, customers, managers, and developers must then establish and agree on the quality goals for each project. The principal goal must be to find and remove all defects in the program at the earliest possible time, with the overall objective of removing all defects before the start of integration and system test. With the goals established, the development teams must make measurable quality plans that can be tracked and assessed to ensure that the project is producing quality work. This in turn requires that the quality of the work be measured at every step, and that the quality data be reviewed and assessed by the developers, their teams, management, and the customer. When defective work is found, it must be promptly fixed. The principle is that defects cost money. The longer they are left in the product, the more work will be built on this defective foundation, and the more it will cost to find and fix them [2].

Step 2: Train and Coach Developers and Teams

Quality work is not done by accident; it takes dedicated effort and properly skilled and motivated professionals. The third principle of software quality is absolutely essential: *The developers must feel personally responsible for the quality of the products they produce.* If they do not, they will not strive to produce quality results, and later trying to find and fix their defects will be costly, time consuming, and ineffective. Convincing developers that quality is their personal responsibility and teaching them the skills required to measure and manage the quality of their work, requires training. While it would be most desirable for them to get this skill and the required knowledge before they graduate from college, practicing software developers must generally learn them from using methods such as the PSP.

With properly trained developers, the development teams then need proper management, leadership, and coaching. Again, the Team Software Process[SM] (TSP[SM]) can provide this guidance and support [3, 4, 5].

Step 3: Manage Requirements Quality

One fundamental truth of all quality programs is that you must start with a quality foundation to have any hope of producing a quality result. In software, requirements are the foundation for everything we do, so the quality of requirements is para-

[SM] Team Software Process and TSP are service marks of Carnegie Mellon University.

mount. However, the requirements quality problem is complicated by two facts.

First, the quality measures must not be abstract characteristics of a requirements document; they should be precise and measurable items such as defect counts from requirements inspections or counts of requirements defects found in system test or customer use. However, to be most helpful, these quality measures must also address the precise understanding the developers themselves have of the requirements regardless of what the requirements originators believe or how good a requirements document has been produced. The developers will build what they believe the requirements say and not what the requirements developers intended to say. This means that the quality-management problem the requirements process must address is the transfer of understanding from the requirements experts to the software developers.

The second key requirements fact is that the requirements are dynamic. As people learn more about what the users need and what the developers can build, their views of what is needed will change. This fact enormously complicates the requirements-management problem. The reason is that people's understanding of their needs evolves gradually and often without any conscious appreciation of how much their views have changed. There is also a time lag: Even when the users know that their needs have changed, it takes time for them to truly understand their new ideas and to communicate them to the developers. Even after the developers understand the changes, they cannot just drop everything and switch to the new version.

To implement a change, the design and implementation implications of every requirements change must be appraised; plans, costs, and commitments adjusted; and agreement reached on how to incorporate this new understanding into the development work. This means that the requirements must be recognized as evolving through a sequence of versions while the development estimates, plans, and commitments are progressing through a similar but delayed sequence of versions. And finally, the product itself will ultimately be produced in a further delayed sequence of versions. The quality management problem concerns managing the quality and maintaining the synchronization of this sequence of parallel requirements, plan, design, and product versions.

Step 4: Statistical Process Control

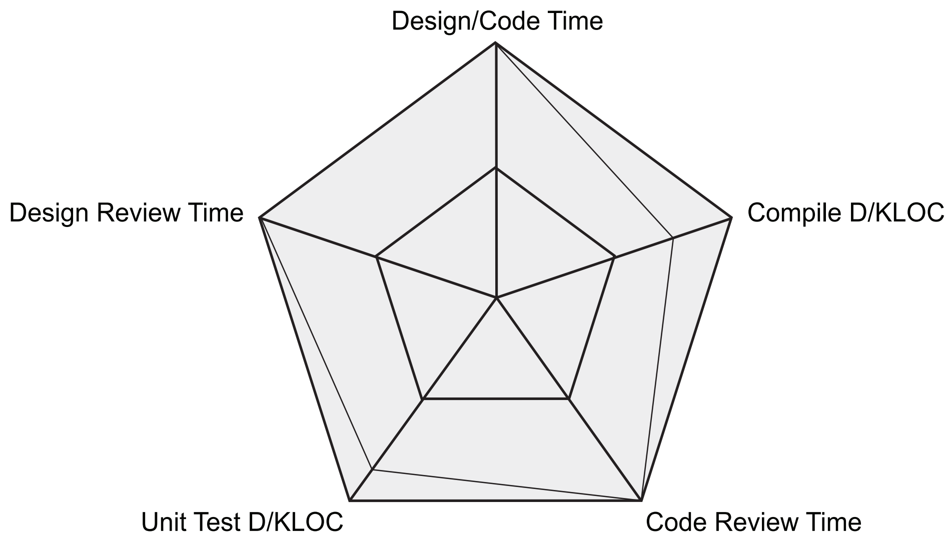
While statistical process control is a large subject, we need only discuss two aspects: process management and continuous process improvement. The first aspect, process management, is discussed here, and process improvement is addressed in Step 8.

The first step in statistical process management is to redefine the quality management strategy. To achieve high levels of software quality, it is necessary to switch from looking for defects to managing the process. As noted earlier, to achieve a quality level of 10 defects per million lines with current software quality management methods, the developers would have to find and fix all but 10 of the 10,000 to 20,000 defects in a program with a 40,000 page listing. Unless someone devises a magic machine that could flawlessly identify every software defect, it would be clearly impossible to improve human search and analysis skills to this degree. Therefore, achieving these quality levels through better testing, reviews, or inspections is not feasible.

A more practical strategy is to measure and manage the quality of the process used to produce the program's parts. If, for example, we could devise a process that would consistently produce 1,000-line modules that each had less than a one percent chance of having a single defect, a system of 1,000 of these modules would likely have less than 10 defects per million lines. One obvious problem with this strategy concerns our ability to devise and properly use such a process.

There has been considerable progress in producing and using such a process. This is accomplished by measuring each developer's process and producing a Process Quality Index (PQI). The TSP quality profile, which forms the basis for the PQI measure, is shown in Figure 5 [6]. Then, the developers and their teams use standard statistical process management techniques to manage the quality of all dimensions of the development work [7]. Data on early TSP teams show that by following this practice, quality is substantially improved [8].

Figure 5: *TSP Software Quality Profile* [6]



Step 5: Quality Evaluation

Quality evaluation has two elements: evaluating the quality of the process used to produce each product element, and evaluating the quality of the products produced by that process. The reason to measure and evaluate process quality, of course, is to guide the process-improvement activities discussed in Step 8. The Capability Maturity Model® Integration (CMMI®) model and appraisal methods were developed to guide process-quality assessments, and the TSP process was developed to guide organizations in defining, using, and improving high-quality processes as well as in measuring, managing, and evaluating product quality.

To evaluate process quality, the developers and their teams must gather data on their work, and then evaluate these data against the goals they established in their quality plan. If any process or process step falls below the team-defined quality threshold, the resulting products must be evaluated for repair, redevelopment, or replacement, and the process must be brought into conformance. These actions must be taken for every process step and especially before releasing any products from development into testing. In product evaluation, the system integration and testing activities are also measured and evaluated to determine if the final product has reached a suitable quality level or if some remedial action is required.

® Capability Maturity Model and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Step 6: Defect Analysis

Perhaps the most important single step in any quality management and improvement system concerns defect data. Every defect found after development, whether by final testing, the users, or any other means must be carefully evaluated and the evaluation results used to improve *both* the process and the product. The reason that these data are so important is that they concern the process failings. Every defect found after development represents a failure of the development process, and each such failure must be analyzed and the results used to make two kinds of improvements.

The first improvement – and the one that requires the most rapid turnaround time – is determining where in the product similar defects could have been made and taking immediate action to find and fix all of those defects. The second improvement activity is to analyze these defects to determine how to prevent similar defects from being injected in the future, and to devise a means to more promptly find and fix all such defects before final testing or release to the user.

Step 7: Configuration Management

For any large-scale development effort, configuration management (CM) is critical. This CM process must cover the product artifacts, the requirements, the design, and the development process. It is also essential to measure and manage the quality of the CM process itself. Since CM processes are relatively standard, however, they need not be discussed further.

Step 8: Process Improvement

The fundamental change required by this software quality-management strategy is to use the well-proven methods of statistical process control to guide continuous process improvement [7]. Here, however, we are not talking about improving the tolerances of machines or the purity of materials; we are talking about managing the quality levels of what people do, as well as the quality levels of their work products. While people will always make mistakes, they tend to make the same mistakes over and over. As a consequence, when developers have data on the defects they personally inject during their work and know how to use these data, they and their teammates can learn how to find just about all of the mistakes that they make. Then, in defining and improving the quality-management process, every developer must use these data to optimally utilize the full range of available defect detection and prevention methods.

Regardless of the quality management methods used (i.e., International Organization for Standardization, correctness-by-construction, or AS9100) continuous improvement strategies such as those defined by CMMI and TSP should be applied to the improvement process itself. This means that the process quality measures, the evaluation methods, and the decision thresholds must also be considered as important aspects of continuous process improvement. Furthermore, since every developer, team, project, and organization is different, it means that this continuous improvement process must involve every person on every development team and on every project in the organization.

Conclusion

While we face a major challenge in improving software quality, we also have substantial and growing quality needs. It should now be clear to just about everyone in the software business that the current testing-based quality strategy has reached a dead end. Software development groups have struggled for years to get quality improvements of 10 to 20 percent by trying different testing strategies and methods, by experimenting with improved testing tools, and by working harder.

The quality improvements required are vast, and such improvements cannot be achieved by merely bulling ahead with the test-based methods of the past. While the methods described in this article have not yet been fully proven for software, we now have a growing body of evidence that they will work – at least better than what we have been doing. What is more, this quality strategy uses the kinds of data-based methods that can guide long-term continuous improvement. In addition to improving quality, this strategy has also been shown to save time and money.

Finally, and most importantly, software quality is an issue that should concern everyone. Poor quality software now costs each of us time and money. In the immediate future, it is also likely to threaten our lives and livelihoods. Every one of us, whether a developer, a manager, or a user, must insist on quality work; it is the only way we will get the kind of software we all need.◆

Acknowledgements

My thanks to Bob Cannon, David Carrington, Tim Chick, Taz Daughtrey, Harry Levinson, Julia Mullaney, Bill Nichols, Bill Peterson, Alan Willett, and Carol Woody for reviewing this article and offering their helpful suggestions. I also much appreciate the constructive suggestions of the CrossTalk editorial board.

References

1. Humphrey, W.S. PSP: A Self-Improvement Process for Software Engineers. Reading, MA: Addison-Wesley, 2005.
2. Jones, C. Software Quality: Analysis and Guidelines for Success. New York: International Thompson Computer Press, 1997.
3. Humphrey, W.S. Winning With Software: An Executive Strategy. Reading, MA: Addison-Wesley, 2002.
4. Humphrey, W.S. TSP: Leading a Development Team. Reading, MA: Addison-Wesley, 2006.
5. Humphrey, W.S. TSP: Coaching Development Teams Reading, MA: Addison-Wesley, 2006.
6. Humphrey, W.S. "Three Dimensions of Process Improvement, Part III: The Team Process" CrossTalk Apr. 1998.
7. Florac, S., and A.D. Carleton. Measuring the Software Process: Statistical Process Control for Software Process Improvement. Reading, MA: Addison Wesley, 1999.
8. Davis, N., and J. Mullaney. "Team Software Process in Practice." SEI Technical Report CMU/SEI-2003-TR-014, Sept. 2003.

About the Author

Watts S. Humphrey joined the Software Engineering Institute (SEI) after his retirement from IBM. He established the SEI's Process Program and led development of the CMM for Software, the PSP, and the TSP. He managed IBM's commercial software development and was vice president of technical development. He is an SEI Fellow, an Association of Computing Machinery member, an Institute of Electrical and Electronics Engineers Fellow, and a past member of the Malcolm Baldrige National Quality Award Board of Examiners. In a recent White House ceremony, the President awarded him the National Medal of Technology. He holds graduate degrees in physics and business administration.

SEI
4500 Fifth AVE
Pittsburgh, PA 15213-2612
Phone: (412) 268-6379
Fax: (412) 268-5758
E-mail: watts@sei.cmu.edu

Coming Events

June 30-July 2
The 17th International Conference on Software Engineering and Data Engineering
Los Angeles, CA
http://sce.cl.uh.edu/sede08/

July 14-17
2008 World Congress in Computer Science, Computer Engineering and Applied Computing Conference
Las Vegas, NV
www.world-academy-of-science.org/worldcomp08/ws/conferences

July 16-17
National Security Space Policy and Architecture Symposium
Chantilly, VA
www.ndia.org

July 20-24
International Symposium on Software Testing and Analysis
Seattle, WA
http://issta08.rutgers.edu

July 27-28
The 44th Annual Aerospace and Defense Contract Management Conference
Garden Grove, CA
www.ncmahq.org/meetings/ADC06

July 28-30
Night Vision Systems
Alexandria, VA
www.iqpc.com/ShowEvent.aspx?id=97070

2009

STC Systems & Software Technology Conference

2009 Systems and Software Technology Conference
Salt Lake City, UT
www.sstc-online.org

Coming Events: **Please submit coming events that are of interest to our readers at least 90 days before registration. E-mail announcements to: nicole.kentta@hill.af.mil.**

Web Sites

Software Quality Profile
www.sei.cmu.edu/publications/articles/quality-profile/index.html
The software community has been slow to use data to measure software quality. This article discusses the reasons for this problem, and describes a way to use process measurements to assess product quality. When the correct data are gathered for every engineer and for every step of the development process, a host of quality measures can be derived to evaluate software quality.

The Software Quality Page
www.swquality.com/users/pustaver/index.shtml
Here is your connection to the world of software quality, standards, and process improvement. The Software Quality Web site contains links to areas including software quality and testing, software inspections and reviews, quality and process metrics, software quality assurance, and other standards, as well as provides helpful links to other software and quality organizations.

American Society for Quality
www.asq.org/pub/sqp/
This site offers articles and discussion on basic concepts, quality tools, organization-wide approaches, people creating quality, using data, specific applications, and other software quality-related issues.

Software Q&A and Testing Resource Center
www.softwareqatest.com
There are many categories of questions and answers when it comes to software quality and assurance testing. This Web site breaks them down into categories that include frequently asked questions, not-so-frequently asked questions, testing resources, test tools, site management tools, jobs, news, and more.

Why Software Quality Matters
www.baselinemag.com/c/a/projects-processes/why-software-quality-matters
As software spreads from computers into auto engines, factory robots, hospital X-ray machines and elsewhere, defects are no longer a problem to be managed. They must be predicted and excised or else unanticipated uses will lead to unintended consequences. This intriguing article proposes innovative solutions to today's emerging software quality issues.

Society for Software Quality
www.ssq.org
The Society for Software Quality (SSQ) is a membership organization for those interested in promoting quality as a universal goal for software. The SSQ promotes increased knowledge and interest in the technology associated with the development and maintenance of quality software.

Better Software Magazine
www.stickyminds.com/bettersoftware/magazine.asp
Better Software is the magazine for software professionals who care about quality. Each issue addresses relevant, timely information to help with building better software. *Better Software* delivers in-depth articles on testing, tools, defect tracking, metrics, and management, and is the only commercial magazine exclusively dedicated to software professionals.

Software Test and Performance Magazine
www.stpmag.com
Software Test & Performance is written for software and application development managers, project managers, team leaders, and test and quality assurance managers. Articles in the magazine provide useful information to help those in the field understand trends and emerging technologies, come to grip with new and timeless challenges, adopt new best practices concepts, and ultimately make better decisions to improve software quality.

Handbook of Software Quality Assurance
www.amazon.com/handbook-software-quality-assurance-3rd/dp/0130104701
The software industry has witnessed a dramatic rise in the impact and effectiveness of software quality assurance. From its infancy when a handful of software pioneers explored the first applications of quality assurance to the development of software, software quality assurance has become integrated into all phases of software development. This handbook capitalizes on the talents and skills of the experts who deal with the implementation of software quality assurance on a daily basis.

Measuring Defect Potentials and Defect Removal Efficiency©

Capers Jones
Software Productivity Research, LLC

There are two measures that have a strong influence on the outcomes of software projects: 1) defect potentials and 2) defect removal efficiency. The term defect potentials refers to the total quantity of bugs or defects that will be found in five software artifacts: requirements, design, code, documents, and bad fixes, *or secondary defects. The term defect removal efficiency refers to the percentage of total defects found and removed before software applications are delivered to customers. As of 2007, the average for defect potentials in the United States was about five defects per function point. The average for defect removal efficiency in the United States was only about 85 percent. The average for delivered defects was about 0.75 defects per function point.*

There are two very important measurements of software quality that are critical to the industry:

1. Defect potentials
2. Defect removal efficiency

All software managers and quality assurance personnel should be familiar with these measurements because they have the largest impact on software quality, cost, and schedule of any known measures.

The phrase *defect potentials* refers to the probable numbers of defects that will be found during the development of software applications. As of 2008, the approximate averages in the United States for defects in five categories, measured in terms of defects per function point and rounded slightly so that the cumulative results are an integer value for consistency with other publications by the author, follow.

Note that defect potentials should be measured with function points and not with lines of code. This is because most of the serious defects are not found in the code itself, but rather in requirements and design. Table 1 shows the averages for defect potentials in the U.S. circa 2008.

The measured range of defect potentials is from just below two defects per function point to about 10 defects per function point. Defect potentials correlate with application size. As application sizes increase, defect potentials also rise.

A useful approximation of the relationship between defect potentials and defect size is a simple rule of thumb: application function points raised to the 1.25 power will yield the approximate defect potential for software applications. Actually, this rule applies primarily to applications developed by organizations at Capability Maturity Model® (CMM®) Level 1. For the higher CMM levels, lower powers would occur. Reference [1] shows additional factors that affect the rule of thumb[1].

The phrase *defect removal efficiency* refers to the percentage of the defect potentials that will be removed before the software application is delivered to its users or customers. As of 2007, the average for defect removal efficiency in the U.S. was about 85 percent.

If the average defect potential is five bugs – or defects – per function point and removal efficiency is 85 percent, then the total number of delivered defects will be about 0.75 per function point. However, some forms of defects are harder to find and remove than others. For example, requirements defects and bad fixes are much more difficult to find and eliminate than coding defects.

At a more granular level, the defect removal efficiency against each of the five defect categories is approximate in Table 2.

Note that the defects discussed in this section include all severity levels, ranging from severity 1: *show stoppers*, down to severity 4: *cosmetic errors*. Obviously, it is important to measure defect severity levels as well as recording numbers of defects[2].

There are large ranges in terms of both defect potentials and defect removal efficiency levels. The *best in class* organizations have defect potentials that are below 2.50 defects per function point coupled with defect removal efficiencies that top 95 percent across the board.

Defect removal efficiency levels peak at about 99.5 percent. In examining data from about 13,000 software projects over a period of 40 years, only two projects had zero defect reports in the first year after release. This is not to say that achieving a defect removal efficiency level of 100 percent is impossible, but it is certainly very rare.

Organizations with defect potentials higher than seven per function point coupled with defect removal efficiency levels of 75 percent or less can be viewed as exhibiting professional malpractice. In other words, their defect prevention and defect removal methods are below acceptable levels for professional software organizations.

Most forms of testing average only about 30 to 35 percent in defect removal efficiency levels and seldom top 50 percent. Formal design and code inspections, on the other hand, often top 85 percent in defect removal efficiency and average about 65 percent.

As can be seen from the short discussions here, measuring defect potentials and defect removal efficiency provide the most effective known ways of evaluating various aspects of software quality control. In general, improving software quality requires two important kinds of process improvement: 1) defect prevention and 2) defect removal.

The phrase *defect prevention* refers to

Table 1: *Averages for Defect Potential*

Requirements defects	1.00
Design defects	1.25
Coding defects	1.75
Documentation defects	0.60
Bad fixes	0.40
Total	**5.00**

Table 2: *Defect Removal Efficiency*

Defect Origin	Defect Potential	Removal Efficiency	Defects Remaining
Requirements defects	1.00	77%	0.23
Design defects	1.25	85%	0.19
Coding defects	1.75	95%	0.09
Documentation defects	0.60	80%	0.12
Bad fixes	0.40	70%	0.12
Total	**5.00**	**85%**	**0.75**

© 2008 Capers Jones. All rights reserved.

® Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

technologies and methodologies that can lower defect potentials or reduce the numbers of bugs that must be eliminated. Examples of defect prevention methods include joint application design, structured design, and also participation in formal inspections[3].

The phrase *defect removal* refers to methods that can either raise the efficiency levels of specific forms of testing or raise the overall cumulative removal efficiency by adding additional kinds of review or test activity. Of course, both approaches are possible at the same time.

In order to achieve a cumulative defect removal efficiency of 95 percent, it is necessary to use approximately the following sequence of at least eight defect removal activities:

- Design inspections.
- Code inspections.
- Unit tests.
- New function tests.
- Regression tests.
- Performance tests.
- System tests.
- External beta tests.

To go above 95 percent, additional removal stages are needed. For example, requirements inspections, test case inspections, and specialized forms of testing, such as human factors testing, add to defect removal efficiency levels.

Since each testing stage will only be about 30 percent efficient, it is not feasible to achieve a defect removal efficiency level of 95 percent by means of testing alone. Formal inspections will not only remove most of the defects before testing begins, it also raises the efficiency level of each test stage. Inspections benefit testing because design inspections provide a more complete and accurate set of specifications from which to construct test cases.

From an economic standpoint, combining formal inspections and formal testing will be cheaper than testing by itself. Inspections and testing in concert will also yield shorter development schedules than testing alone. This is because when testing starts after inspections, almost 85 percent of the defects will already be gone. Therefore, testing schedules will be shortened by more than 45 percent.

When IBM applied formal inspections to a large database project, delivered defects were reduced by more than 50 percent from previous releases, and the overall schedule was shortened by about 15 percent. Testing itself was reduced from two shifts over a 60-day period to one shift over a 40-day period. More importantly, customer satisfaction improved to *good* from prior releases where customer satisfaction previously had been very poor.

Measurement of Defect Potentials and Defect Removal Efficiency

Measuring defect potentials and defect removal efficiency levels are among the easiest forms of software measurement, and are also the most important. To measure defect potentials it is necessary to keep accurate records of all defects found during the development cycle, which is something that should be done as a matter of course. The only difficulty is that *private* forms of defect removal such as unit testing will need to be done on a volunteer basis.

Measuring the numbers of defects found during reviews, inspections, and testing is also straightforward. To complete the calculations for defect removal efficiency, customer-reported defect reports submitted during a fixed time period are compared against the internal defects found by the development team. The normal time period for calculating defect removal efficiency is 90 days after release.

As an example, if the development and testing teams found 900 defects before release, and customers reported 100 defects in the first three months of usage, it is apparent that the defect removal efficiency would be 90 percent.

Unfortunately, although measurements of defect potentials and defect removal efficiency levels should be carried out by 100 percent of software organizations, the frequency of these measurements circa 2008 is only about five percent of U.S. companies. In fact, more than half of U.S. companies do not have any useful quality metrics at all. More than 80 percent of U.S. companies, including the great majority of commercial software vendors, have only marginal quality control and are much lower than the optimal 95 percent defect removal efficiency level. This fact is one of the reasons why so many software projects fail completely or experience massive cost and schedule overruns. Usually failing projects seem to be ahead of schedule until testing starts, at which point huge volumes of unanticipated defects stop progress almost completely.

As it happens, projects that average about 95 percent in cumulative defect removal efficiency tend to be optimal in several respects. They have the shortest development schedules, the lowest development costs, the highest levels of customer satisfaction, and the highest levels of team morale. This is why measures of defect potentials and defect removal efficiency levels are important to the industry as a whole; these measures have the greatest impact on software performance of any known metrics.

Additionally, as an organization progresses from the U.S. average of 85 percent in defect removal efficiency up to 95 percent, the saved money and shortened development schedules result because most schedule delays and cost overruns are due to excessive defect volumes during testing. However, to climb above 95 percent defect removal efficiency up to 99 percent does require additional costs. It will be necessary to perform 100 percent inspections of every deliverable, and testing will require about 20 percent more test cases than normal.

It is an interesting sociological observation that measurements tend to change human behavior. Therefore, it is important to select measurements that will cause behavioral changes in positive and beneficial directions. Measuring defect potentials and defect removal efficiency levels have been noted to make very beneficial improvements in software development practices.

When these measures were introduced into large corporations such as IBM and ITT, in less than four years the volumes of delivered defects had declined by more than 50 percent, maintenance costs were reduced by more than 40 percent, and development schedules were shortened by more than 15 percent. There are no other measurements that can yield such positive benefits in such a short time span. Both customer satisfaction and employee morale improved, too, as a direct result of the reduction in defect potentials and the increase in defect removal efficiency levels.◆

Reference

1. Jones, Capers. Estimating Software Costs. 2nd edition. McGraw-Hill, New York: 2007.

Notes

1. The averages for defect potentials are derived from studies of about 600 companies and 13,000 projects. Non-disclosure agreements prevent the identification of most companies. However some companies such as IBM and ITT have provided data on defect potentials and removal efficiency levels.
2. The normal period for measuring defect removal efficiency starts with requirements inspections and ends 90 days after delivery of the software to its users or customers. Of course, there

are still latent defects in the software that will not be found in 90 days, but having a 90-day interval provides a standard benchmark for defect removal efficiency. It might be thought that extending the period from 90 days to six months or 12 months would provide more accurate results; however, updates and new releases usually come out after 90 days, so these would dilute the original defect counts. Latent defects found after the 90-day period can exist for years, but on average about 50 percent of residual latent defects are found each year. The results vary with number of users of the applications. The more users, the faster residual latent defects are discovered.
3. Formal design and code inspections are the most effective defect removal activity in the history of software, and are also very good in terms of defect prevention. Once participants in inspections observe various kinds of defects in the materials being inspected, they tend to avoid those defects in their own work. All software projects larger than 1,000 function points should use formal design and code inspections.

Additional Reading

1. Boehm, Barry W. Software Engineering Economics. Prentice Hall, Englewood Cliffs, NJ; 1981.
2. Crosby, Philip B. Quality Is Free. New American Library, Mentor Books. New York: 1979.
3. Garmus, David, and David Herron. Function Point Analysis. Addison Wesley Longman, Boston: 2001.
4. Garmus, David, and David Herron. Measuring the Software Process: A Practical Guide to Functional Measurement. Prentice Hall, Englewood Cliffs, NJ: 1995.
5. Grady, Robert B., and Deborah L. Caswell. Software Metrics: Establishing a Company-Wide Program. Prentice-Hall: 1987.
6. International Function Point Users Group. IT Measurement. Addison Wesley Longman, Boston: 2002.
7. Jones, Capers. Applied Software Measurement. 3rd edition; McGraw-Hill, New York: 2008.
8. Jones, Capers. “Sizing Up Software.” Scientific American New York: Dec. 1998.
9. Jones, Capers. Software Assessments, Benchmarks, and Best Practices. Addison Wesley Longman. Boston: 2000.
10. Jones, Capers. Conflict and Litigation Between Software Clients and Developers. Software Productivity Research, Burlington, MA: 2003.
11. Kan, Stephen H. Metrics and Models in Software Quality Engineering. 2nd edition. Addison Wesley Longman, Boston: 2003.

About the Author

Capers Jones is currently the president of Capers Jones and Associates, LLC. He is also the founder and former chairman of Software Productivity Research (SPR) where he holds the title of Chief Scientist Emeritus. He is a well-known author and international public speaker, and has authored the books “Patterns of Software Systems Failure and Success,” “Applied Software Measurement,” “Software Quality: Analysis and Guidelines for Success,” “Estimating Software Costs,” and “Software Assessments, Benchmarks, and Best Practices.” Jones and his colleagues from SPR have collected historical data from more than 600 corporations and more than 30 government organizations. This historical data is a key resource for judging the effectiveness of software process improvement methods.

Software Productivity Research, LLC
Phone: (877) 570-5459
Fax: (877) 570-5459
E-mail: cjonesiii@cs.com

Quality Processes Yield Quality Products

Thomas D. Neff
MTC Technologies

Would your company like to save $100,000 per day? Would you like to surge an urgent project's delivery time by 50 percent and deliver zero errors? Software organizations have done just that. In this article, I list small steps you can take that will lead your company toward similar results based on my 15 years of process improvement experience.

If your company developed software that ran tools capable of propelling big objects long distances, measured accuracy in miles, and increased its accuracy to inches, you might save your customer millions of dollars. This actually happened [1]. If your company refined its software development processes so that your unit testing department found zero errors in a three-year period, you might eliminate unit testing and move those testers into other types of testing, saving many dollars with every release. This also happened [2]. If your customer asked you to speed up your next software delivery by 50 percent and *guarantee no flaws* in the delivered product, could you do it without incurring any extra costs or sacrificing other projects? One company did [3].

You may figure those goals are impossible for your organization to achieve or you do not have enough money to make it happen. If so, you are wrong. Right now open a Web browser, type <www.sei.cmu.edu>, and hit enter. If your organization does software development, search for Capability Maturity Model Integration for Development (CMMI-DEV). If you are an acquisition organization, search for CMMI-Acquisition (ACQ). All organizations should check out People Capability Maturity Model (P-CMM).

These models are all instantiations of Total Quality Management (TQM), the method that turned low-quality Japanese trinkets into high-quality automobiles, electronics, cameras, and many other products [4]. Because these models are different views of the same paradigm, you can also use them in other areas. One company used a predecessor of the CMMI-DEV and achieved the model's highest level of process maturity in software development. That company's hardware people realized the software folks really had their act together. They got jealous and sought the software secret. When they were shown the CMM, they said, "We could use that if we just change a few of the terms. Instead of talking about managing software requirements, we would manage hardware requirements." Before long, that entire group was achieving record low manufacturing defects, record high profits, record high customer and employee satisfaction, record low employee turnover, and many more positive effects [3].

If the rewards from doing this are so great, why do so few companies achieve CMMI Level 5 – the highest level of process maturity? I contend it is because they do not execute their continuous process improvement (CPI) effort properly. There are many ways to do it right, but even more ways to do it wrong. If you would like to help ensure success in your CPI effort, read this article and get started. Before long you could very well be producing (or acquiring) software of exceptional quality, precisely meeting customer requirements, and incurring minimal maintenance costs.

Based on 15 years of CPI experience, here are some items you might consider when starting or reinvigorating your CPI effort. While they are no guarantee that you will reach the CMMI pinnacle, they can help you avoid pitfalls that snag many such efforts. (Throughout this list, *we* and *our* refer to the Nuclear Weapons Effects Division Process Improvement Team [PIT]):

- **Do not try to inspect in quality.** All too often, people believe they can have ad-hoc development processes, then use an inspection process *at the end* and effectively remove all defects, yielding a quality product. It just will not happen. My experience shows that only a small portion of defects are actually removed if the attempt is only at the end. Inspections in every phase of the process are good, just do not wait to the end and then do a lone inspection! Industry statistics indicate that for every four errors pulled out, one new error is injected. Hence, you must iterate many times to approach zero. Large expense, little return – not a good business decision.
- **Do not look for a quick fix.** I have learned to fear when a senior manager goes to a conference where process improvement is discussed. Inevitably they return with the latest fad and want to implement it by week's end. It takes between two and three years to get CPI institutionalized. Your processes did not get screwed up overnight; they will not get fixed overnight, either.
- **Hold people accountable.** This is the biggest key to any CPI effort. If you create a meager CPI plan complete with a feedback loop for improvements, then hold people accountable to following it – you will make great progress in relatively little time. I have experienced both sides of this and can vouch that *not* holding them accountable will guarantee failure, and always holding them accountable is more likely to guarantee success. However, you cannot hold them accountable for six months and then give up because it is not working. Refer to the second point above.
- **Do not aim for a certain level of improvement.** Never state, "We want to achieve CMMI Level 3 by ___ date." What matters are the qualities exhibited, not the score obtained. Your primary emphasis must be to institutionalize CPI. Once that is accomplished, the rest will fall into place. If your aim is Level 3, once you reach it, you will not have any objective left and you will begin backsliding. However, if you emphasize CPI, once you reach Level 5, you will be thinking about what Level 6 (if there was one) would look like or you will seek other company areas that could benefit from your CPI attitude. Levels are just indicators of your progress.
- **Do not follow the CMMIs in the order they are written.** They are written so that *one size fits all.* As you and I know, even though one size fits all, it rarely looks good. You are much better off finding those areas of the CMMI currently giving you the most headaches and work on those first. If that does not work for you or you have many headaches, take a new project

and use it to pilot the CMMI. As you work through that project, write the necessary standard operating instructions (SOI)/standard operating procedures (SOPs), as identified by the CMMI, and test them with that project. Once they are acceptable, publish them as an example of how your organization does business. Of course, these are living documents and as you mature, your processes must evolve with you.

- **Perfect is the enemy of good enough.** If you are looking to produce perfect processes, you will never get there. Aim for the 80 percent solution. While that might seem pretty low, remember that each process has a feedback loop whereby improvements can easily and frequently be made. I know of no one who has ever gone to work thinking, "I want to do worse today than yesterday." Most employees want to do a better job. The problem is that they do not always know how, but processes can give them a framework. Their experience and intuition will help fill in the details on how to improve.
- **Jealousy is a great thing.** Do not let the lack of senior management support stop you. All you need is *any* manager to support CPI to get it going. Once you are making progress, others will see something is different with your manager: projects are being produced on time, on budget, and/or with greater quality. The other managers will become jealous and want to achieve the same success.
- **Do not try to end world hunger.** Aim low and reach your target. If you try to fix your whole company, you will likely spend most of your time negotiating, selling, and/or compromising. It is better to fix your little niche and make others jealous (see above). Allow them to modify your processes to fit their needs. They will already have incentive to ensure they are successful (jealousy still reigns). If they fail, they will come back since they will become jealous over something else you are doing better than them (and reaping tangible rewards such as a bigger budget or additional people). They might even stumble onto a better process than yours – great! Ask to use it and make it work for you. Now you have a strong ally.
- **Two heads are better than one.** Create a PIT encompassing each work unit in your area. As a PIT leader, you do not have the corner on good ideas. The more people you include (up to seven, plus or minus two), the better. However, you do not want just anyone. You want people who share your enthusiasm for process improvement and who see the big picture. If you have the wrong team members, it can be detrimental because you will spend 80 percent of your time educating 20 percent of them.
- **Every team needs cheerleaders.** If you document/improve all processes but no one knows about them, you have accomplished nothing. Find the opinion leaders in each work area and get them involved. If they are not on the PIT, try to include them on the occasional work group or have them write an article for the PIT newsletter. That newsletter can be another good cheerleader. It is your first opportunity to provide training snippets on new processes as well as keeping everyone informed of your current CPI status.

"One company used a predecessor of the CMMI-DEV and achieved the model's highest level of process maturity in software development. That company's hardware people realized the software folks had their act together ... When they were shown the CMM, they said, "We could use that ....""

- **Fail and get over it.** As humans we are imperfect. Do not worry about failure. The only failure is one where you learn nothing. If you are unsuccessful and learn from it then it was a great learning experience, not a failure because you now know at least one way *not* to do it.
- **Start with the obvious.** The CMMI-ACQ has a lot of information about what your acquisition program should contain. It does not tell you *how* to do it but there is plenty of *what*. Do not wait until you have the *how* to get started. Take the *what* (i.e., CMMI) and turn it into a policy statement (SOI). Then, when it is time to create the *how* (SOP), people will know which *how* to develop. You will have already added some structure to your process improvement effort.
- **Keep focused.** When writing an SOI, do not delve into *how* people should do something. You want to focus on *what* they are to do and, on occasion, *why*. You can even describe a little of *who* or *when*. Once an SOP is created then it is time to describe *how* to do the job. These SOIs/SOPs should not be written for a three-year-old, but they also should not be written for a brain surgeon (unless it is an SOP on brain surgery). You should rarely include any *why* material in an SOP. If the worker does not know *why* they are doing their job, they have bigger issues. SOPs should be written by those already doing the job.
- **We don't need no stinkin' tools.** Just as everyone wants instant gratification, we also want a super tool to make our jobs easier, thus solving all of our problems. Come back to reality. That tool does not exist. I have found that if you buy a tool to solve your problems, you are more likely to get a failed CPI effort – and be poorer to boot. Because you do not have a repeatable process, the tool only lets you make mistakes faster, easier, and with greater impact. Of course, this frustrates people and they will quit using the tool. They will not realize it was the lack of process that caused the problems, not the tool. First, create a process and *then* introduce a tool to help people perform the process faster, cheaper, and better.
- **Sometimes status quo is good.** Remember, people want to do their jobs better – they just do not want to change to do that. The mere act of documenting your current (probably flawed) processes is a huge improvement over undocumented processes. At least now you *could* repeat the process twice in a row. It is better to get the early buy-in than try to perfect the process too quickly. There will be plenty of opportunity to improve the process as people use it.
- **Sometimes status quo is bad.** Hopefully you will never hear *we do it this way because it is how we have always done it.* However, someone is thinking

it. My experience is that if people do not know why they are doing something, they are also ripe for the suggestion that there might be a better way, especially if it means less work. Many of the *always done it this way* processes can be reduced in effort by 50 percent or more. Often, some work products are used by no one. If a product has no customer (user), eliminate it. You will earn many new friends. If there was a hidden customer, they will eventually figure out something changed and come to you to explain why they need what you eliminated. Then you will know why it is needed.

- **Keep it short.** We keep SOIs to no more than three pages. Most are one to one-and-a-half pages, with the shortest being two paragraphs. SOPs are longer, but we still try to keep them to about four pages. If attachments are added, we do not count those against the four-page goal. A short document will get read, but a long one will not. Our plan is to write 100 short documents instead of one all-encompassing volume.
- **Do not get hung up on training.** Some people feel they need training on everything. At some level, I agree. However, it is just as bad to do too much training as not enough. No one *needs* training on our SOIs. Even most SOPs are written so that anyone sufficiently educated could pick up an SOP and determine how to perform its task. Use screen captures, pictures, and flowcharts – some people like words,some need pictures. Cater to both but keep it short, and provide training as needed or requested.
- **A hyperlink is your friend.** Ample hyperlinking avoids redundancy and inaccuracy. For instance, we have an SOI describing acronyms and definitions. All acronyms and definitions used in our SOIs/SOPs are included here. We then name the definition as a bookmark and hyperlink upon its first use in each document. That way we ensure the proper definition is used and we do not have to spell it out, which keeps our documents shorter.
- **Procrastination is your enemy.** There is no bad place to start a CPI effort except to not start at all. I do not know how many times I have been asked how to start a CPI effort. I answer, "It doesn't matter." Start where you feel most comfortable, with what causes the most headaches, with what will give the best return on investment, or you can use any other criteria. As Nike said, *just do it.*
- **I think I can, I think I can.** The Little Engine That Could ran uphill for a long time. It was about out of steam when it crested the hill and things became easier. So it is with CPI. You *will* face an uphill battle for at least six months – and probably more. However, at some point (that point will be different for each organization) you will crest the hill and gain momentum. At that point, no one can stop your CPI effort. It will be institutionalized and no longer dependent on individuals, becoming an integral part of your organization's business practices. As long as you have steam, you *must* keep chugging uphill. Set your sights just over the crest and you will get there.
- **This is not three-card monte. Pick a model, any model.** There are many process models from which to choose (CMM, CMMI, International Organization for Standardization [ISO] 9000, TQM, Lean, Six Sigma, Lean Six Sigma, etc.). Which should you use? When you are getting started, it does not matter. Just pick one and go. Any improvement is better than none. You may even choose bits and pieces of several models. Having said that, I believe the CMM and CMMI models are the most comprehensive and take you farther than the others. For instance, ISO 9000 takes you to about a CMMI Level 2. The Lean and Six Sigma models require you to document your process first, so you can determine just how much it has improved. Since most organizations just starting CPI do not have documented processes, it seems the CMM/CMMI might be best for starting because they provide guidance on what should be in your key processes. As your processes mature, you will likely incorporate other models into your CPI effort to speed your progress or improve the quality. Use whatever works for you.
- **Do not reinvent the wheel.** *Reuse* is your friend. Build on others' successes. Learn from them. Never embrace *not invented here* syndrome. The Software Engineering Institute already developed all the tools you need to start making significant leaps in the quality of your processes. Their CMM and CMMI models describe every characteristic your organization should exhibit at various levels of process maturity. Use the models – they work. They will give you quality processes leading to quality products.

From what I have seen, most failed CPI efforts could not figure out where to begin, lost steam before starting, could not get any management support (usually tried at too high a level), focused too much on tools versus processes, could not find a quick fix and quit, or tried to solve world hunger and gave up.

Based on my 15 years in process improvement, I suspect that if you follow these suggestions, sticking with it at least two years, you will be successful in your CPI effort.

If you have CPI lessons learned, I would enjoy hearing them.◆

References

1. Yamamura, George, and Gary B. Wigle. "SEI CMM Level 5: For the Right Reasons," CrossTalk Aug. 1997 <www.stsc.hill.af.mil/crosstalk/frames.asp?uri=1997/08/seicmm5.asp>.
2. Billings, C., J. Clifton, B. Kolkhorst, E. Lee, and W.B. Wingert. "Journey to a Mature Software Process." IBM Systems Journal. Vol. 33, No. 1. 1994: pp. 46-61.
3. Vu, John D. "Presentation to CIO's Office." National Reconnaissance Office. Chantilly, VA., Mar 2001.
4. Deming, W. Edwards. Out of the Crisis. MIT Press. 1986.

About the Author

Thomas D. Neff, Lt Col, U.S. Air Force (Ret.) spent most of his Air Force career in software development and project management. Currently, he works for MTC Technologies supporting the Defense Threat Reduction Agency's Nuclear Weapons Effects Division as a process manager, and uses the CMMI-ACQ and P-CMM as guides for that effort. Neff is a frequent speaker on process improvement at information technology conferences. He has a Master of Computer Science from Texas A&M, which helped steer him toward process improvement.

DTRA/RD-NTE
8725 JJ Kingman RD
Ft Belvoir, VA 22060-6201
Phone: (703) 767-4106
Fax: (703) 767-9844
E-mail: thomas.neff_contractor @dtra.mil

Urgent Reader Request!

Have we helped? How?

CROSSTALK has been there for you for almost twenty years, and now we are asking that you be there for CROSSTALK. As a free publication, your comments are the lifeblood of our existence. Has the information provided in our publication ever helped you save time or money? Have you benefitted in other ways? If so, we want to hear about it. Our goal has always been to inform and educate you — our readers — on software engineering best practices, processes, policies and other technologies. If we have succeeded in this goal, let us know "how, where, when, and why." Your comments will help CROSSTALK continue to bring you the news and information you've come to expect.

Send your stories of success to **crosstalk.publisher@hill.af.mil**, or go to **www.stsc.hill.af.mil/crosstalk**.
We will feature your comments in our 20th anniversary issue this August.
Thank You!

Articles ... and metrics helped me save $33.5 million on government programs.

Quoting it all the time to substantiate project plans and estimates.

... Between CrossTalk and our tech advisor, we shipped 580K + SLOC with more functionality than originally planned.

Share Your Results!

The Use and Limitations of Static-Analysis Tools to Improve Software Quality

Dr. Paul Anderson
GrammaTech, Inc.

Advanced static-analysis tools have been found to be effective at finding defects that jeopardize system safety and security. This article describes how these work and outlines their limitations. They are best used in combination with traditional dynamic testing techniques, and can even reduce the cost to create and manage test cases for stringent run-time coverage.

Static analysis has commonly been known as a technique for finding violations of superficial stylistic programming rules, and for alerting programmers to typing discrepancies in type-unsafe languages. The latest static-analysis tools go far beyond this, and are capable of finding serious errors in programs such as null-pointer de-references, buffer overruns, race conditions, resource leaks, and other errors. They can do so without requiring additional input from the users, and without requiring changes to development processes or practices. Actionable results are produced quickly with a low level of false positives. These static-analysis tools are not a silver bullet, however, because they can never prove that a program is completely free of flaws. The following is a description of how static-analysis tools work, followed by a discussion of how they can be used to complement traditional testing.

How Static Analysis Finds Flaws

The first thing a static analysis tool must do is identify the code to be analyzed. The source files that must be compiled to create a program may be scattered across many directories, and may be mixed in with other source code that is not used for that program. Static analysis tools operate much like compilers so they must be able to identify exactly which source files contribute and should ignore those that do not. The scripts or build system that builds the executable obviously know which files to use, so the best static analysis tools can extract this information by reading those scripts directly or by observing the build system in action. This way the tool gets to see not only the source files but also which compiler is being used and any command-line flags that were passed in. The parser that the static analysis tool uses must interpret the source code in the same way that the real compiler does. It does this by modeling how the real compile works as closely as possible. The command-line flags are an essential input to that.

As the build system progresses, each invocation of the compiler is used to create a whole program model of the program. This model consists of a set of abstract representations of the source, and is similar to what a compiler might generate as an intermediate representation. It includes the control-flow graph, the call graph, and information about symbols such as variables and type names.

“In order to understand the limitations of the techniques that these tools use, it is important to understand the metrics used to assess their performance.”

Once the model has been created, the analysis performs a symbolic execution on it. This can be thought of as a simulation of a real execution. Whereas a real execution would use concrete values in variables, the symbolic execution uses abstract values instead. This execution explores paths and, as it proceeds, if any anomalies are observed, they are reported as warnings. This approach is based on abstract interpretation [1] and model checking [2].

The analysis is *path-sensitive*, which means that it can compute properties of individual paths through the program. This is important because it means that when a warning is reported, the tool can tell the user the path along which execution must proceed in order for the flaw to be manifest. Tools also usually indicate the points along that path where relevant transformations occur and conditions on the data values that must hold. These help users understand the result and how to correct the problem should it be confirmed.

Once a set of warnings have been issued, these tools offer features to help the user manage the results, including allowing the user to manually label individual warnings. Warnings that correspond to real flaws can be labeled as true positives. Warnings that are false alarms can be labeled as false positives. Warnings that are technically true positives but which are benign can be labeled as *don't care*. Most tools offer features that allow the user to suppress reporting of such warnings in subsequent analyses.

Limitations of Static Analysis

In order to understand the limitations of the techniques that these tools use, it is important to understand the metrics used to assess their performance. The first metric, *recall*, is a measure of the ability of the tool to find real problems. Recall is measured as the number of flaws found divided by all flaws present. The second metric is *precision*, which measures the ability of the tool to exclude false positives. It is the ratio of true positives to all warnings reported. The third metric is *performance*. Although not formally defined, this is a measure of the computing resources needed to generate the results.

These three metrics usually operate in opposition to each other. It is easy to create a tool that has perfect precision and excellent performance – one that reports no lines contain flaws will satisfy because it reports no false positives. Similarly, it is easy to create a tool with perfect recall and excellent performance – one that reports that all lines have errors will answer because it reports no false negatives. Clearly, however, neither tool is of any use whatsoever.

Finally, it is at least theoretically possible to write an analyzer that would have excellent precision and excellent recall given enough time and access to enough processing power. Whether such a tool would be as useless as the previous two example tools is debatable and would depend on just how much time it would take. What is clear is that no such tools currently exist and to create them would be very difficult.

As a result, all tools occupy a middle ground around a sweet spot that developers find most useful. Developers expect analyses to complete in time roughly proportional to the size of their code base and within hours rather than days. Tools that take longer simply do not get used because they take too long. Low precision means more false positives, which has an insidious effect on users. As precision goes down, even true positive warnings are more likely to be erroneously judged as false positives because the users lose trust in the tool.

For most classes of flaws, precision less than 80 percent is unacceptable. For more serious flaws, however, precision as low as five percent may be acceptable if the code is to be deployed in very risky environments. It is difficult to quantify acceptable values for recall as it is impossible to measure accurately in practice, but clearly users would not bother using these tools at all if they did not find serious flaws that escape detection by other means.

Each of these constraints introduces its own set of limitations, however they are all interrelated. The reasons that lead to low recall are explained in more detail in the following sections.

Path Limitations

As mentioned earlier, these analyses are path sensitive. This improves both recall and precision and is probably the key aspect of these products that makes them most useful. A full exploration of all paths through the program would be very expensive. If there are *n* branch points in a procedure, and there are no loops in that procedure, then the number of intraprocedural paths through that procedure can be as many as 2^n. In practice, this is fewer because some branches are correlated, but the asymptotic behavior remains. If procedure calls and returns are taken into account, the number of paths is *doubly* exponential, and if loops are taken into account then the number of paths is unbounded. Clearly it is not possible for a tool to explore all of these paths. The tools restrict their exploration in two ways. First, loops are handled by exploring a small fixed number of iterations: often, the first time around the loop is singled out as special, and all other iterations are considered en masse and represented by an approximation. Second, not all paths are explored. It is typical for an analysis to place an upper bound on the number of paths explored in a particular procedure or on the amount of time available, and a selection of those remaining paths are explored.

"If asynchronous paths can occur (such as those caused by interrupts or exceptions) or if the program uses concurrency, then the number of possible paths to consider increases further. Many tools simply ignore the possibilities."

If asynchronous paths can occur (such as those caused by interrupts or exceptions) or if the program uses concurrency, then the number of possible paths to consider increases further. Many tools simply ignore these possibilities. Finally, most tools also ignore recursive function calls, and function calls that are made through function pointers (or make very coarse approximations) as considering these also contributes to poor performance and poor precision.

Abstract Domain

As previously mentioned, these tools work by exploring paths and looking for anomalies in the abstract state of the program. The appeal of the symbolic execution is that each abstract state represents potentially many possible concrete states. For example, given an 8-bit variable x, there are 2^8 possible concrete values: 0, 1, …, 255. The symbolic execution, however, might represent the value as two abstract states: $x=0$, and $x>0$. So where a concrete execution has 256 states to explore, the symbolic execution has only two.

As such, the expressivity of this abstract domain is an important factor that determines the effectiveness of the analysis. Again, there is a trade-off here: better precision and recall can be achieved by more sophisticated abstract domains, but more resources will then be required to complete an analysis. Values in the abstract domain are equations that represent constraints on values, i.e., $x=0$, or $y>10$. As the analysis progresses, a constraint solver is used to combine and simplify these equations. A key characteristic of these abstract domains is that there is a special value, usually named *bottom*, which indicates that the analysis knows no useful information about the actual value. *Bottom* is the abstract value that corresponds to all possible concrete values. Reaching bottom is impossible to avoid for any non-trivial abstraction in general as this would require solving the halting problem. Once bottom is reached, the analysis has a choice of treating it as a potentially dangerous value, which would increase recall, or as a probably safe value, which would increase precision. Most tools opt for the latter as the former also has the effect of decreasing precision enormously.

If there are program constructs that step outside the bounds of what can be expressed in the abstract domain, this causes the analysis to lose track of variables and their relationships. For example, an abstract domain that allows the expression of affine relationships between no more than two variables admits expressions such as $x=2y$. However, something such as $x=y+z$ is out of bounds because it involves three variables and the analysis would be forced to conclude $x=bottom$ instead.

The consequence of this is the abstract domain that a tool uses determines a great deal about the kind of flaws that it is capable of detecting. For example, if the tool uses an abstract domain of affine relations between two variables, then it may fail to find flaws that depend on three variables.

```
if (a || b || c)
      x = 0;
```

Coverage	a	b	c
Statement	T	-	-
Decision	T	-	-
	F	F	F
MCDC	T	-	-
	F	T	-
	F	F	T
	F	F	F

Table 1: *Test Cases Needed for Statement, Decision, and MCDC Coverage*

Similarly, most tools choose a domain that allows them to reason about the values of integers and addresses but not floating-point values, so they will fail to find flaws in floating-point arithmetic (such as divide by zero).

Missing Source Code

If the source code to a part of a program is not available, as is almost always the case because of operating system and third-party libraries, or if the code is written in a language not recognized by the analysis tool, then the analysis must make some assumptions about how that missing code operates. Take, for example, a call to a function in a third-party library that takes a single pointer-typed parameter and returns an integer. In the absence of any other information, most analyses will assume that the function does nothing and returns an unknown value. This clearly is not realistic, but it is not practical to do better in general. The function may de-reference its pointer parameter, it may read or write any global variable that is in scope, it may return an integer from a particular range, or it may even abort execution. If the analysis knew this, it would have better precision and recall but it is forced to make the simple assumption unless told otherwise.

There are two approaches around this. First, if source is not available but object code is, then the analysis could be extended into the object code. This is a highly attractive solution but no products are available yet. The technological basis for such a tool exists, however [3], and it is expected that products capable of analyzing object code as well as C/C++ will appear.

A second approach to the problem is to specify stubs, or *models*, that summarize key aspects of the missing source code. The popular analysis tools provide models for commonly used libraries such as the C library. These models only have to approximate the behavior of the code. Users can, of course, write these themselves for their own libraries but it can be a tricky and time-consuming effort.

Out of Scope

There are, of course, entire classes of flaws that static analysis is unlikely ever to be able to detect. Static analysis excels at finding places where the fundamental rules of the language are being violated such as buffer overruns, or where commonly used libraries are being used incorrectly, or where there are inconsistencies in the code that indicate misunderstanding. If the code does the wrong thing for some other reason, but does not then terminate abnormally, then static analysis is unlikely to be able to help because it is unable to divine the intent of the author. For example, if a function is intended to sort in ascending order, but perfectly sorts in descending order instead, then static analysis will not help much. This kind of functionality testing is what traditional dynamic testing is good for.

Static Analysis and Testing

Static analysis should never be seriously considered as a replacement for traditional dynamic testing activities. Rather, it should be thought of as a way of amplifying the software assurance effort. The cheapest bug to find is the one that gets found earliest, and as static analysis can be used very early in the development cycle, its use can reduce the cost of development and liberate resources for use elsewhere. This is the traditional view of how static analysis can reduce testing costs. However, there is a second way in which the use of static analysis can reduce the cost of testing: it makes it easier to achieve full coverage.

One measure of the effectiveness of a test suite is how well it exercises or *covers* the code being tested. There are many different kinds of coverage. Statement coverage is the most common, but for riskier code more stringent forms are often required. Decision coverage is a superset of statement coverage, and requires that all branches in the control flow of the program are taken. In DO-178B, a development standard for flight software [4], the riskiest code is required to be tested with 100 percent modified condition/decision coverage (MCDC). This means that a test suite must be chosen such that all subexpressions in all conditionals are evaluated to both true and false. Table 1 illustrates how many different test cases are needed for each to achieve coverage. For the code sample on the left, the values required of the boolean variables a, b, and c to achieve each form of coverage is shown on the right.

Achieving full coverage, even for statement coverage, can be very time consuming. The engineer creating the test case must figure out what inputs must be given to drive the program to each statement. What can make it very frustrating is if it is fundamentally impossible to do so, but this may not be

Figure 1: *A Redundant Condition Warning*

```
                 c:\CodeSonar\ex2.c
                 Enter foo
                   void foo (int rest, int length)
                 {
                     if (rest <=1)
                         buf[pos-1] = '>';
                     else if (rest == 2)
                         buf[pos++] = '>';
                     else if (length > rest)
Always True:             if (--rest > 1) {    /* Redundant Condition (ID: 1) */
rest > 1
                             if (rest >= 2)
                                 rest --;
                         }
```

Figure 2: *A Second Redundant Condition Warning*

```
Never True:           if (!flags & MASK)     /*Redundant Condition */
($temp2 & 16) != 0
                      {
                          error("Cannot sign packet");
                          return;
                      }
```

apparent simply by looking at the code. If the program contains unreachable code, then statement coverage is impossible. If it contains redundant conditions (those that are either always true or always false), then MCDC is impossible. Developers can spend hours trying to refine a test case before it is evident that their efforts are pointless.

If the unreachable code or redundant conditions can be brought to the attention of the tester early, then they do not need to waste time in a futile attempt to achieve the impossible. This is what static analysis can do easily and efficiently. Figure 1 shows an example of a report from CodeSonar[1] illustrating a redundant condition in a sample of code taken from an open-source application. The variable rest, an unaliased integer, must be at least three by line 12. The decrement on that line means it is at least two, so the condition will always be true. The following line is also redundant and shown in a different report.

In this example, all the components of the code relevant to the redundancy are in close proximity so it is likely that a reviewer would have spotted this during a manual review. It would not have been so easy to spot if the code were more complex. If the code had spanned several pages, or if relevant parts had been embedded in function calls or macro invocations, then it would have been difficult to spot. Static analysis is not sensitive to superficial aspects of the code such as its layout, so it would not have been confused.

These kinds of redundancies correlate well with genuine flaws as well; for example, consider the example in Figure 2. This was distilled from a genuine flaw found in a widely used open-source program, and is a redundant condition warning where the tool has deduced that the true branch of the conditional will never be taken. The reason why it concluded so is shown to the left. The first operand to the bitwise AND (the `&` symbol) is either zero or one as this is the range of the negation operator (the `!` symbol). This is what is represented by `$temp2`. The constant `MASK` has the value 16. The result of the AND expressions `1&16` and `0&16` are both zero, so the conditional expression is guaranteed to be zero.

The programmer who wrote this code probably misunderstood the precedence of the operators in the conditional expression and assumed that the innermost operator had higher precedence. If so, then a correction would be to place parentheses around the inner expression. This is a potentially dangerous flaw as it means that the error condition would not be detected, which could result in unpredictable behavior.

When to Use Static Analysis Tools

The best time to use advanced static analysis tools is early in the development cycle. In Holzmann's 10 rules for safety-critical development [5], the most far-reaching rule states that these tools should be used throughout the development process. As well as reducing the cost of development by finding flaws earlier and reducing testing effort, early adoption exerts a force on programmers to write code that is more amenable to analysis, thereby increasing the probability that the tool will find errors. Care should be taken, however, to avoid a risk compensation phenomenon, where programmers use less care because they assume that the static analysis tool will find their mistakes.

If adopted late in the development cycle, static analysis may issue a large number of warnings. The best value is gained if these are all dealt with, either by fixing the code, marking them as false positives, or labeling them as *don't care* if they are believed to be benign. However, if scheduling time to sift through these is not feasible, then an alternative strategy is to operate in a *differential* mode, where programmers are only told about new warnings. This way they are alerted to flaws in code that they are working with while it remains fresh in their minds.

Conclusion

Advanced static analysis tools offer much to help improve the quality of software. The best tools are easy to integrate into the development cycle, and can yield high-quality results quickly without requiring additional engineering effort. They can be used not just for finding flaws, but also to guide testing activities. They use sophisticated symbolic execution techniques for which engineering trade-offs have been made so that they can generate useful results in a reasonable time. As such, they inevitably have both false positives and false negatives, and so should never be considered a replacement for traditional testing techniques.◆

References

1. Cousot, P., and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." ACM Symposium on Principles of Programming Languages. Los Angeles, CA., 1977.
2. Clarke, E.M., O. Grumberg, and D.A. Peled. Model Checking. MIT Press: Cambridge, MA: 1999.
3. Balakrishnan, G., R. Gruian, T. Reps, and T. Teitelbaum. "CodeSurfer/x86 – A Platform for Analyzing x86 Executables." International Conference on Compiler Construction. 2005.
4. RTCA/DO-178B. "Software Considerations in Airborne Systems and Equipment Certification." 1992.
5. Holzmann, G.J. "The Power of 10: Rules for Developing Safety-Critical Code." IEEE Computer 2006.

Note

1. GrammaTech's static analysis tool.

About the Author

Paul Anderson, Ph.D., is vice president of engineering at GrammaTech, a spin-off of Cornell University that specializes in static analysis, where he manages GrammaTech's engineering team and is the architect of the company's static analysis tools. He has worked in the software industry for 16 years, with most of his experience focused on developing static analysis, automated testing, and program transformation tools. A significant portion of Anderson's work has involved applying program analysis to improve security. His research on static analysis tools and techniques has been reported in numerous articles, journal publications, book chapters, and international conferences. Anderson has a B.Sc. from Kings College, University of London, and his doctorate in computer science from City University, London.

GrammaTech, Inc.
317 N Aurora ST
Ithaca, NY 14850
Phone: (607) 273-7340
Fax: (607) 273-8752
E-mail: paul@grammatech.com

Automated Combinatorial Test Methods – Beyond Pairwise Testing

D. Richard Kuhn and Dr. Raghu Kacker
National Institute of Standards and Technology

Dr. Yu Lei
University of Texas, Arlington

Pairwise testing has become a popular approach to software quality assurance because it often provides effective error detection at low cost. However, pairwise (2-way) coverage is not sufficient for assurance of mission-critical software. Combinatorial testing beyond pairwise is rarely used because good algorithms have not been available for complex combinations such as 3-way, 4-way, or more. In addition, significantly more tests are required for combinations beyond pairwise testing, and testers must determine expected results for each set of inputs. This article introduces new tools for automating the production of complete test cases covering up to 6-way combinations.

Many testers are familiar with the most basic form of combinatorial testing – *all pairs* or pairwise testing, in which all possible pairs of parameter values are covered by at least one test [1, 2]. Pairwise testing uses specially constructed test sets that guarantee testing every parameter value interacting with every other parameter value at least once. For example, suppose we had an application that is intended to run on a variety of platforms comprised of five components: an operating system (Windows XP, Apple OS X, Red Hat Linux), a browser (Internet Explorer, Firefox), protocol stack (IPv4, IPv6), a processor (Intel, AMD), and a database (MySQL, Sybase, Oracle), a total of 3 x 2 x 2 x 2 x 2 = 48 possible platforms. With only 10 tests, as shown in Figure 1, it is possible to test every component interacting with every other component at least once, i.e., all possible pairs of platform components. The effectiveness of pairwise testing is based on the observation that software faults often involve interactions between parameters. While some bugs can be detected with a single parameter value, such as a divide-by-zero error, the toughest bugs often can only be detected when multiple conditions are true simultaneously. For example, a router may be observed to fail only for the User Datagram Protocol (UDP) when packet rate exceeds 1.3 million packets per second – a 2-way interaction between protocol type and packet rate. An even more difficult bug might be one which is detected only for UDP when packet volume exceeds 1.3 million packets per second and packet chaining is used – a 3-way interaction between protocol type, packet rate, and chaining option.

Unfortunately, only a handful of tools can generate more complex combinations, such as 3-way, 4-way, or more (we refer to the number of variables in combinations as the *combinatorial interaction strength*, or simply, interaction strength, e.g., a 4-way combination has 4 variables and thus its interaction strength is 4). The few tools that do generate tests with interaction strengths higher than 2-way may require several days to generate tests [3] because the generation process is mathematically complex. Pairwise testing, i.e. testing 2-way combinations, has come to be accepted as the standard approach to combinatorial testing because it is computationally tractable and can effectively detect many faults. For example, pairwise testing could detect 70 percent to more than 90 percent of software faults for the applications studied in [4].

But if pairwise testing can detect 90 percent of bugs, what interaction strength is needed to detect 100 percent? Surprisingly, we found no evidence that this question had been studied when the National Institute of Standards and Technology (NIST) began investigating software faults in 1996. Results showed that across a variety of domains, all failures could be triggered by a maximum of 4-way to 6-way interactions [5]. As shown in Figure 2, the detection rate increases rapidly with interaction strength. With the NASA application, for example, 67 percent of the failures were triggered by only a single parameter value, 93 percent by 2-way combinations, and 98 percent by 3-way combinations. The detection rate curves for the other applications are similar, reaching 100 percent detection with 4-way to 6-way interactions. That is, six or fewer variables were involved in all failures for the applications studied, so 6-way testing could, in theory, detect all of the failures. While not conclusive, these results suggest that combinatorial testing that exercises high strength interaction combinations can be an effective approach to high-integrity software assurance.

Applying combinatorial testing to real-world software presents a number of challenges. For one of the best algorithms, the number of tests needed for combinatorial coverage of n parameters with v values each is proportional to $v^t \log n$, where t is the interaction strength [3]. Unit testing of a small module with 12 parameters required only a few dozen tests for 2-way combinations, but approximately 12,000 for 6-way combinations [6]. But a large number of test cases will not be a barrier if they can be produced with little human intervention, thus reducing cost. To apply combinatorial testing, it is necessary to find a set of test inputs that covers all t-way combinations of parameter values, and to match up each set of inputs with the expected output for these input values.

Figure 1: *Pairwise Test Configurations*

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	IE	IPv4	Intel	Sybase
6	OS X	Firefox	IPv4	Intel	Oracle
7	RHL	IE	IPv6	AMD	MySQL
8	RHL	Firefox	IPv4	Intel	Sybase
9	RHL	Firefox	IPv4	AMD	Oracle
10	OS X	Firefox	IPv6	AMD	Oracle

These are both difficult problems, but they can now be solved with new algorithms on currently available hardware. We explain these two steps followed by a small but complete illustrative example.

Computing T-Way Combinations of Input Values Using FireEye

The first step in combinatorial testing is to find a set of tests that will cover all *t*-way combinations of parameter values for the desired combinatorial interaction strength *t*. This collection of tests is known as a *covering array*. The covering array specifies test data where each row of the array can be regarded as a set of parameter values for an individual test. Collectively, the rows of the array cover all *t*-way combinations of parameter values. An example is given in Figure 3, which shows a 3-way covering array for 10 variables with two values each. The interesting property of this array is that any three columns contain all eight possible values for three binary variables. For example, taking columns F, G, and H, we can see that all eight possible 3-way combinations (000, 001, 010, 011, 100, 101, 110, 111) occur somewhere in the rows of the three columns. In fact, this is true for any three columns. Collectively, therefore, this set of tests will exercise all 3-way combinations of input values in only 13 tests, as compared with 1,024 for exhaustive coverage. Similar arrays can be generated to cover up to all 6-way combinations. A non-commercial research tool called FireEye [3], developed by NIST and the University of Texas at Arlington[1], makes this possible with much greater efficiency than previous tools. For example, a commercial tool required 5,400 seconds to produce a less-optimal test set than FireEye generated in 4.2 seconds.

Matching Combinatorial Inputs With Expected Outputs Using Nu Symbolic Model Verifier (SMV)

The second step in combinatorial test development is to determine what output should be produced by the system under test for each set of input parameter values, often referred to as the *oracle problem* in testing. The conventional approach to this problem is human intervention to design tests and assign expected results or, in some cases, to use a *reference implementation* that is known to be correct (for example, in checking conformance of various vendor products to a protocol standard). Because combinatorial testing can require a large number of tests, an automated method is needed for determining the expected results for each set of input data. To solve this problem, we use the open-source NuSMV model checker [7] (an enhanced version of the well-known SMV model checker [7]). Conceptually, the model checker can be viewed as exploring all states of a system model to determine if a property claimed in a specification statement is true. What makes a model checker particularly valuable is that if the claim is false, the model checker not only reports this, but also provides a *counterexample* showing how the claim can be shown false. As will be seen in the illustrative example, this gives us the ability to match every set of input test data with the result that the system should produce for that input data. Figure 4 outlines the process.

The model checker thus automates the work that normally must be done by a human tester – determining what the correct output should be for each set of input data. Other approaches to determining the correct output for each test can also be used. For example, in some cases we can run a model checker in simulation mode, producing expected results directly rather than through a counterexample, but the approach illustrated in this article is more general, and can be applied to non-deterministic systems or used with mutation-based methods in addition to combinatorial testing [8]. The method chosen for resolving the oracle problem depends on the problem at hand, but model checking can be effective in testing protocols, access control, or other applications where there is a state machine, unified modeling language state chart, or other formal model available.

Illustrative Example

Here we present a small example of an access control system. The rules of the system are a simplified multi-level security system, followed by a step-by-step construction of tests using an automated process. Each subject (user) has a clearance level *u_l*, and each file has a classification level *f_l*.

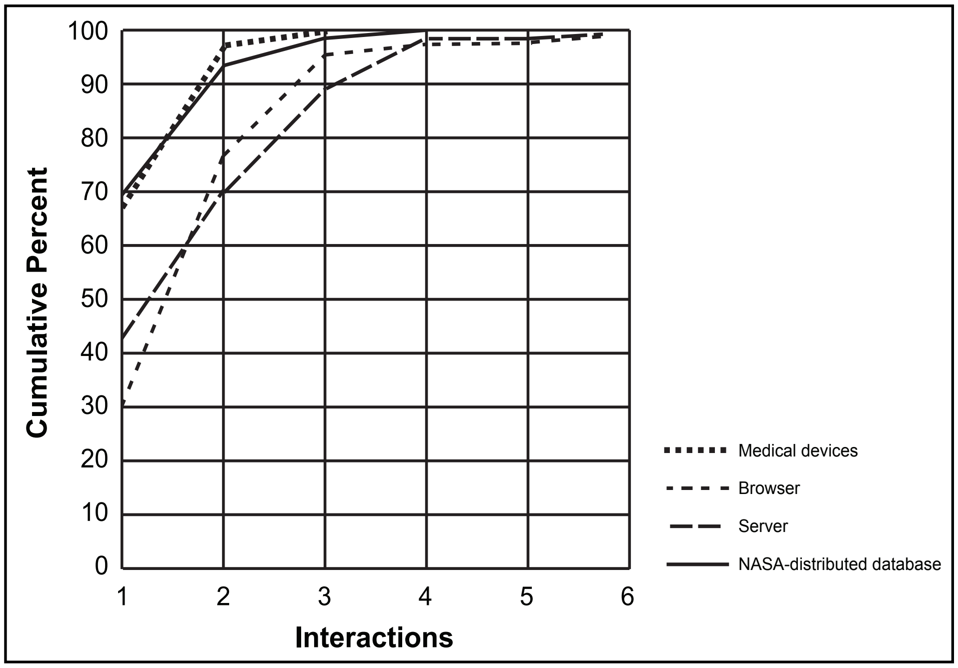


Figure 2: *Error Detection Rates for Interaction Strengths 1 to 6*

Figure 3: *3-way Covering Array for 10 Parameters With Two Values Each*

A	B	C	D	E	F	G	H	I	J
0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

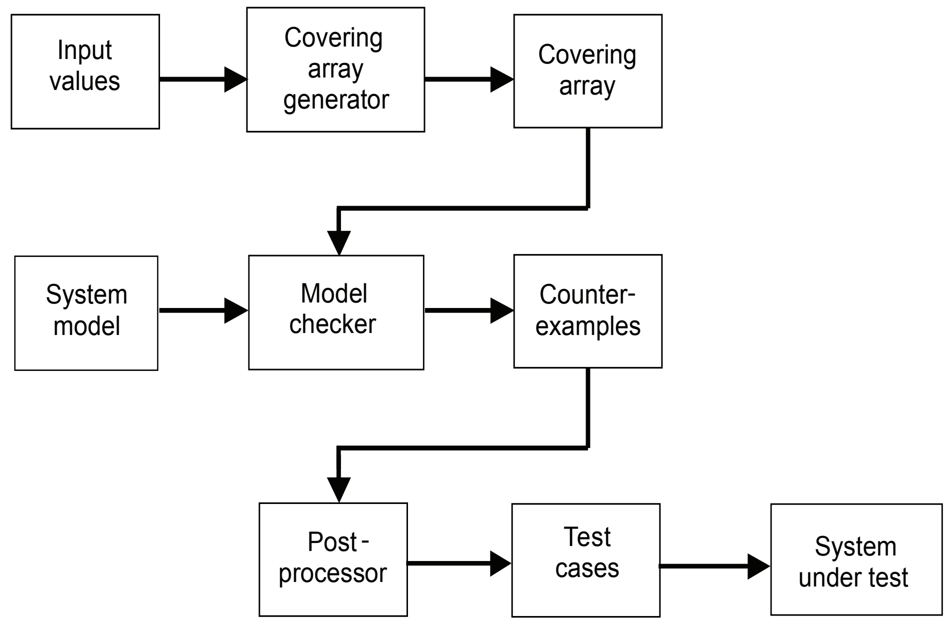


Figure 4: *Automated Combinatorial Test Construction*

Levels are given as 0, 1, or 2, which could represent levels such as Confidential, Secret, and Top Secret. A user *u* can read a file *f* if *u_l* ≥ *f_l* (the *no read up* rule), or write to a file if *f_l* ≥ *u_l* (the *no write down* rule).

Thus, a pseudo-code representation of the access control policy is:

```
if u_l >= f_l & act = rd then
   GRANT;
else if f_l >= u_l & act = wr
   then GRANT; else DENY;
```

Tests produced will check that these rules are correctly implemented in a system.

Figure 5: *SMV Model of Access Control Rules*

```
   1. MODULE main
   2. VAR
        --Input parameters
   3. u_l:   0..2;          -- user level
   4. f_l:   0..2;          -- file level
   5. act:  {rd, wr};       -- action

        --output parameter
   6. access: {START_, GRANT,DENY};

   7. ASSIGN
   8. init(access) := START_;
        --if access is allowed under rules, then next state is GRANT
        --else next state is DENY
   9. next(access) := case
   10.     u_l >= f_l & act = rd : GRANT;
   11.     f_l >= u_l & act = wr : GRANT;
   12.     1 : DENY;
   13.     esac;

   14.     next(u_l) := u_l;
   15.     next(f_l) := f_l;
   16.     next(act) := act;

-- reflection of the assigns for access
-- if user level is at or above file level then read is OK
SPEC AG ((u_l >= f_l & act = rd ) -> AX (access = GRANT));

-- if user level is at or below file level, then write is OK
SPEC AG ((f_l >= u_l & act = wr ) -> AX (access = GRANT));

-- if neither condition above is true, then DENY any action
SPEC AG (!( (u_l >= f_l & act = rd ) | (f_l >= u_l & act = wr ))
         -> AX (access = DENY));
```

System Model

This system is easily modeled in the language of the NuSMV model checker as a simple two-state finite state machine. Other tools could be used, but we illustrate the test production procedure using NuSMV because it is among the most widely used model checkers and is freely available. Our approach is to model the system as a simple state machine, then use NuSMV to evaluate the model and post-process the results into complete test cases.

Figure 5 shows the system model defined in SMV. The `START` state initializes the system (line 8), with the rule noted previously used to evaluate access as either `GRANT` or `DENY` (lines 9-13). For example, line 10 represents the first line of the pseudo-code example: in the current state, (always `START` for this simple model), if *u_l* ≥ *f_l* then the next state is `GRANT`. Each line of the case statement is examined sequentially, as in a conventional programming language. Line 12 implements the *`else DENY`* rule, since the predicate *1* is always true. `SPEC` clauses given at the end of the model define statements that are to be proven or disproven by the model checker. The SPEC statements in Figure 5 duplicate the access control rules as temporal logic statements and are, thus, provable. In the following sections, we illustrate how to combine them with input data values to generate complete tests with expected results.

In SMV, specifications of the form `AG (predicate 1) -> AX (predicate 2)` indicate essentially that for all paths (the `A` in `AG`) for all states globally (the `G`), if `predicate 1` holds then (->) for all paths, in the next state (the `X` in `AX`) `predicate` *2* will hold. SMV checks the properties in the `SPEC` statements and shows that they match the access control rules as implemented in the finite state machine, as expected. Once the model is correct and `SPEC` claims have been shown valid for the model, counterexamples can be produced that will be turned into test cases.

Generating Covering Array

We will compute covering arrays that give all *t*-way combinations, with degree of interaction coverage two for this example. If we had a larger number of parameters, we would produce test configurations that cover all 3-way, 4-way, etc., combinations. (With only three parameters, 3-way interaction would be equivalent to exhaustive testing, so we use 2-way combinations for illustration purposes.) The first step is to define the parameters (using the graphical user interface if desired) and their values in a system definition file that will be used as input to the covering array generator FireEye with the following format: After the system definition file is saved, we run FireEye, in this case specifying 2-way interactions. FireEye produces the output shown in Figure 6.

Each test configuration defines a set of values for the input parameters `u_l`, `f_l`, and `act`. The complete test set ensures that all 2-way combinations of parameter values have been covered

Model Claims With Covering Array Values Inserted

The next step is to assign values from the covering array to parameters used in the model. For each test, we write a claim that the expected result will not occur. The model checker determines combinations that would disprove these claims, outputting these as counterexamples. Each counterexample can then be converted to a test with known expected result. For example, for Test 1 the parameter values are:

```
u_l = 0 & f_l = 0 & act = rd
```

For each of the nine configurations in the covering array (Figure 7), we create a `SPEC` claim of the form: `SPEC AG`*(covering array values)* -> `AX` *!(access = result).*

This process is repeated for each possible result, in this case either *GRANT* or *DENY*, so we have nine claims for each of the two results. The model checker is able to determine, using the model defined previously, which result is the correct one for each set of input values, producing a total of nine tests.

Excerpt:

```
SPEC AG((u_l = 0 & f_l = 0 & act
   = rd) -> AX !(access = GRANT));
SPEC AG((u_l = 0 & f_l = 1 & act
   = wr) -> AX !(access = GRANT));
SPEC AG((u_l = 0 & f_l = 2 & act
   = rd) -> AX !(access = GRANT));
etc.

SPEC AG((u_l = 0 & f_l = 0 & act
   = rd) -> AX !(access = DENY));
SPEC AG((u_l = 0 & f_l = 1 & act
   = wr) -> AX !(access = DENY));
SPEC AG((u_l = 0 & f_l = 2 & act
   = rd) -> AX !(access = DENY));
etc.
```

Generating Counterexamples With Model Checker

NuSMV produces counterexamples where the input values would disprove the claims specified in the previous section. Each of these counterexamples is, thus, a set of test data that would have the expected result of `GRANT` or `DENY.` For each SPEC claim, if this set of values cannot in fact lead to the particular result, the model checker indicates that this is true. For example, for the configuration below, the claim that access will not be granted is true, because the user's clearance level (`u_l = 0`) is below the file's level (`f_l = 2`):

```
-- specification AG (((u_l
= 0 & f_l = 2) & act = rd)
-> AX !(access = GRANT)) is
true
```

If the claim is false, the model checker indicates this and provides a trace of parameter input values and states that will prove it is false. In effect, this is a complete test case, i.e., a set of parameter values and an expected result. It is then simple to map these values into complete test cases in the syntax needed for the system under test. An excerpt from NuSMV output is shown in Figure 8.

The model checker finds that six of the input parameter configurations produce a result of GRANT and three produce a DENY result, so at the completion of this step we have successfully matched up each input parameter configuration with the result that should be produced by the system under test.

At first, the method previously described may seem *backward*. Instead of negating each possible result, why not simply produce tests from model checker output such as `specification AG (((u_l = 0 & f_l = 2) & act = rd) -> AX (access = DENY))` *is true*? Such a procedure would work fine for this simple example, but more sophisticated testing may require more information. Note that if the claim is true, the model checker simply reports the fact while if it is false, a trace of inputs and internal states is produced to show how the claim fails. Some testing may require information on internal states or variable values, and the previous procedure provides this information.

Shell Script Post-Processing to Produce Complete Tests

The last step is to use a post-processing tool that reads the output of the model checker and generates a set of test inputs with expected results. The post-processor strips out the parameter names and values, giving tests that can be applied to the system under test. Simple scripts are then used to convert the test cases into input for a suitable test harness. The tests produced are shown in Figure 9 (see next page).

Conclusion

While tests for this trivial example could easily have been constructed manually, the procedures introduced in this tutorial can – and have – been used to produce tens of thousands of complete test cases in a few minutes once the SMV model

```
u_l: 0,1,2
f_l: 0,1,2
act: rd, wr
```

Figure 6: *Model Parameters and Values*

Test	u_l	f_l	act
1	0	0	rd
2	0	1	wr
3	0	2	rd
4	1	0	wr
5	1	1	rd
6	1	2	wr
7	2	0	rd
8	2	1	wr
9	2	2	wr

Figure 7: *FireEye Output Test Values*

Figure 8: *Counterexamples (excerpt)*

```
-- specification AG (((u_l = 0 & f_l = 0) & act = rd)
       -> AX !(access = GRANT))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  u_l = 0
  f_l = 0
  act = rd
  access = START_
-> Input: 1.2 <-`
-> State: 1.2 <-
  access = GRANT
...
etc.
```

```
u_l = 0 & f_l = 0 & act = rd -> access = GRANT
u_l = 0 & f_l = 1 & act = wr -> access = GRANT
u_l = 1 & f_l = 1 & act = rd -> access = GRANT
u_l = 1 & f_l = 2 & act = wr -> access = GRANT
u_l = 2 & f_l = 0 & act = rd -> access = GRANT
u_l = 2 & f_l = 2 & act = rd -> access = GRANT
u_l = 0 & f_l = 2 & act = rd -> access = DENY
u_l = 1 & f_l = 0 & act = wr -> access = DENY
u_l = 2 & f_l = 1 & act = wr -> access = DENY
```

Figure 9: *Test Cases*

has been defined for the system under test. The methods in this article still require human intervention and engineering judgment to define a formal model of the system under test and for determining appropriate abstractions and equivalence classes for input parameters. But by automating test generation we can provide much more thorough testing than is possible with most conventional methods. In addition, the testing has a sound empirical basis in the observation that software failures have been shown to be caused by the interaction of relatively few variables. By testing all variable interactions to an appropriate strength, we can provide stronger assurance for critical software.◆

References

1. Daich, G.T. "New Spreadsheet Tool Helps Determine Minimal Set of Test Parameter Combinations." CrossTalk Aug. 2003.
2. Phadke, M.S. "Planning Efficient Software Tests." CrossTalk Oct. 1997.
3. Lei, Y., R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence. "IPOG/IPOG-D: Efficient Test Generation for Multi-Way Combinatorial Testing." Software Testing, Verification, and Reliability (to appear 2008).
4. Kuhn, D.R., D. Wallace, and A. Gallo. "Software Fault Interactions and Implications for Software Testing." IEEE Transactions on Software Engineering 30(6):418-421, 2004.
5. Wallace, D.R., and D.R. Kuhn. "Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data." International Journal of Reliability, Quality and Safety Engineering 8(4):351-371, 2001.
6. Kuhn, D.R., and V. Okun. "Pseudo-Exhaustive Testing for Software." Proc. of 30th NASA/IEEE Software Engineering Workshop. Apr. 2006.
7. Cimatti, A., E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. "NuSMV 2: An OpenSource Tool for Symbolic Model Checking." Proc. of International Conference on Computer-Aided Verification. Copenhagen, Denmark.
8. Ammann, P., and P.E. Black. "Abstracting Formal Specifications to Generate Software Tests via Model Checking." Proc. of 18th Digital Avionics Systems Conference. St. Louis, MO. Oct. 1999.

Notes

1. Available on <http://csrc.nist.gov/acts>.
2. The tool can be downloaded at <http://nusmv.irst.itc.it/>. More information on SMV can be found at <www.cs.cmu.edu/~modelcheck/>.

About the Authors

D. Richard Kuhn is a computer scientist in the computer security division of the National Institute of Standards and Technology (NIST). His primary technical interests are in information security, software assurance, and empirical studies of software failure. He co-developed the role based access control model (RBAC) used throughout industry, and led the effort to establish RBAC as an American National Standards Institute standard. Kuhn has a masters degree in computer science from the University of Maryland, College Park, and a bachelors and master of business administration from William & Mary.

NIST
MS 8930
Gaithersburg, MD 20899-8930
Phone: (301) 975-3337
Fax: (301) 975-8387
E-mail: kuhn@nist.gov

Yu Lei, Ph.D., is an assistant professor of computer science at the University of Texas, Arlington. He was a member of the Fujitsu Network Communications, Inc., technical staff from 1998 to 2001. Lei's research is in the area of automated software analysis, testing, and verification. His current research is supported by NIST. Lei has a bachelor's degree from Wuhan University, a master's degree from Chinese Academy of Sciences, and a doctorate from North Carolina State University.

The University of Texas at Arlington
Department of Computer Science and Engineering
P.O. Box 19015
Arlington, TX 76019-0015
Phone: (817) 272-2341
Fax: (817) 272-3784
E-mail: ylei@cse.uta.edu

Raghu Kacker, Ph.D., is a mathematical statistician in the mathematical and computational sciences division of the NIST. His current interests include software testing, uncertainty in physical and virtual measurements, interlaboratory evaluations, and Bayesian uncertainty in measurement. Kacker received his doctorate in statistics from Iowa State University.

NIST
100 Bureau DR
MS 8910
Gaithersburg, MD 20899-8910
Phone: (301) 975-2109
Fax: (301) 975-3553
E-mail: raghu.kacker@nist.gov

Software Quality Unpeeled

Dr. Jeffrey Voas
SAIC

The expression software quality *has many interpretations and meanings. In this article, I do not attempt to select any one in particular, but instead help the reader see the underlying considerations that underscore software quality. Software quality is a lot more than standards, metrics, models, testing, etc. This article digs into the mystique behind this elusive area.*

The term software quality has been one of the most overused, misused, and overloaded terms in software engineering. It is a generic term that suggests quality software but lacks general consensus on meaning. Attempts have been made to define it. The Institute for Electronics and Electrical Engineers (IEEE) Standard 729 defines it as:

> … totality of features of a software product that bears on its ability to satisfy given needs and … composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer. [1]

However, this attempt and others are few, and not precise. In fact the second edition of the Encyclopedia of Software Engineering [2] does not have it listed as an entry; the encyclopedia skips straight from "Software Productivity Consortium" to "software reading." And worse, books with software quality in the title never give a definition to it in their pages [3, 4].

If you review the past 20 years or so, you will find an abundance of other terms that have been employed as pseudo-synonyms for software quality. Examples include process improvement, software testing, quality management, the International Organization for Standardization 9001, software metrics, software reliability, quality modeling, configuration management, Capability Maturity Model® Integration, benchmarking, etc. In doing so, the term *software quality* has wound up representing a family of processes and ideas more than it represents *good enough* software. In short, software quality has become a culture and community more than a technical goal [5].

In this article, I will avoid the quicksand associated with trying to come up with a one-size-fits-all definition. Instead, I will expose how software quality is composed of various layers and, by peeling off different layers, it allows us to have a rational discussion between a typical software supplier and end user such that an agreement can be reached as to whether or not the software is good enough.

Certification

We will begin dissecting software quality by first looking at the multiple viewpoints behind the term *certification*. This will provide us with a look into our first layer.

"*In some instances phantom users more heavily determine whether the software is fit for purpose than the traditional inputs. In short, it is environment that gives fit for purpose context.*"

The term is often used to refer to certifying people skills. For example, the American Society for Quality (ASQ) has a host of certifications that individuals can attain in order to demonstrate competence in certain fields, e.g., they can become an ASQ Certified Software Quality Engineer. An individual can also become certified in specific commercial software packages, e.g., a Microsoft Certified Software Engineer.

For the purposes here, I employ a different perspective that comes from three schools of thought. The first school deals with certifying that a certain set of development, testing, or other processes applied during the pre-release phases of the life-cycle were satisfied. In doing so, you certify that the *processes were followed and completed*. (Demonstrating that they were applied correctly is a trickier issue.) In the second school, you certify that the developed software *meets the functional requirements*; this can be accomplished via various types of testing or other analyses. For the third school, you can certify that the software itself is *fit for purpose*. This third school will be the most useful, and throughout this article, it will be considered software to be *good enough* if it is fit for purpose.

In this article, the term *purpose* suggests that two things are present: (1) executable software; and (2) an operating environment. An *environment* is a complex entity: It involves the set of inputs that the software will receive during execution, along with the probability that the events will occur [6]. This is referred to as the *operational profile* [6]. But it also involves the hardware that the software operates on, the operating system, available memory, disk space, drivers, and other background processes that are potentially competing for hardware resources, etc. These other factors are as much a part of the environment as are the traditional inputs; they have been termed *invisible* or *phantom* users.

In some instances phantom users more heavily determine whether the software is fit for purpose than the traditional inputs. In short, it is environment that gives fit for purpose context. By more completely defining and thus bounding the environment to include phantom users, we gain an advantage in that we can reduce the set of assumptions needed to predict whether the software is good enough. Understanding the distinction between traditional inputs and phantom users is one ingredient needed to argue that *fit for purpose* has been achieved.

Further, note that rarely will there be only one environment that software, and in particular general purpose software, will encounter during operation. That offers a key insight as to why general purpose software is not certified by independent laboratories; such laboratories could not be

Scenario	Meets Requirements	Satisfies Development Processes	Fit for Purpose
1	No	No	No
2	No	No	**Yes**
3	No	Yes	No
4	No	Yes	**Yes**
5	Yes	No	No
6	Yes	No	**Yes**
7	Yes	Yes	No
8	Yes	Yes	**Yes**

Table 1: *Views on Certification*

omnipotent and could not know all of the potential target environments [7].

By revisiting the three schools of thought on certification, we discover eight ways to visualize software quality (See Table 1). Let us look at a couple.

In Table 1, scenario 2 represents a system that did not meet the requirements and was not developed according to the specified development procedures, but miraculously, the end result was software that was usable in the field. While this seems implausible, it is possible. Scenario 7 is the opposite: a system that met the requirements and was developed according to specified development procedures but resulted in unusable software. Scenario 7 may seem like heresy to many in the community of software quality practitioners. It is not; it simply dispels the myth that requirements elicitation is far from a perfect science and that simply following common sense *do's* and *dont's* (as spelled out in a development process plan) guarantees good enough software [8].

Note that only four of these scenarios yield good enough software: 2, 4, 6, and 8. The other four provide a product that is not usable for its target environment and that brings us back to the discussion of why scoping the target environment as precisely as possible is an important piece of what software quality means.

In summary, *fit for purpose* is the nearest of the three certification schools of thought of the IEEE definitions for software quality. However, we cannot only rely on knowing the environment and expect to be justified in proclaiming we have achieved software quality. Let us explore other considerations.

Three High-Level Attributes of Fit for Purpose

Most readers would probably be comfortable with labeling software as being of good quality if the software could ensure that (1) it produces accurate and reliable output, (2) it produces the needed output in a timely manner, and (3) it produces the output in a secure and private manner. These three criteria simply state that you get the right results at the right time at the correct level of security. These are the next three considerations that cannot be ignored and must incorporate into what software quality means.

While each of these is intuitive, none is precise enough. The family of attributes referred to as the ilities is a good starting point to help increase that precision [9]. This family includes behavioral characteristics such as reliability, performance, safety, security, availability, fault-tolerance, etc. (These attributes are also sometimes termed non-functional requirements.) Other family members such as dependability, survivability, sustainability, testability, interoperability, and scalability each require some degree of one or more of the first six attributes. So, for example, to have a dependable system, some level of reliable and fault-tolerant behavior is necessary. To have a survivable system, some amount of fault tolerance and availability is required, and so on. However, to simplify this discussion towards our goal of understanding the term software quality, we will focus only on the first six: reliability, performance, safety, security, availability, and fault-tolerance.

"Ility" Oxymorons

Table 2 illustrates combinations of these six *ilities*. If we were to flesh this table out as we did in Table 1 we would have 64 rows; however, here we only show 14 combinations for brevity. (For the cells left empty, we are not considering the degree to which that attribute contributes to the quality of the software's behavior.)

Let us look at a few of these categories and determine what they represent:

- **Category 1.** Suggests that the software is reliable, has good performance, does not trigger unsafe events to occur (e.g., in a transportation control system), has appropriate levels of security built in, has good availability and thus does not suffer from frequent failures resulting in downtime, and is resilient to internal failures (i.e., fault tolerant). Is all of this possible in a single software system?
- **Category 2.** Suggests that the software offers reliable behavior, but suffers from the likelihood of producing outputs that send the system that the software feeds inputs into an unsafe mode. This would represent a safety-critical system where hazardous fail-

Table 2: *Ility Combinations*

Category	Reliability	Performance	Safety	Security	Availability	Fault-tolerance
1	Yes	Yes	Yes	Yes	Yes	Yes
2	Yes		No			
3	No		Yes			
4	No			Yes		
5	No				Yes	
6		No			Yes	
7		Yes	No			
8			Yes	No		
9			No	Yes		
10	Yes			No		
11	Yes			No	Yes	
12	Yes	No		Yes		
13		Yes		Yes		
14	No	No	No	No	No	No

ures are unacceptable; hazardous failures are categorized differently for such systems than failures that do not facilitate possible disastrous loss-of-life or loss-of-property consequences. (Note that software by itself is never unsafe; however software is often referred to as unsafe if it produces outputs to a system that put the system into an unsafe mode. Safety is a system property, not a software property.) A classic example of a reliable product that is unsafe is placing a functioning toaster into a bathtub of water with the cord still connected; the toaster is reliable, but it is not safe to go near.

- **Category 3.** Suggests that the software behaves so unreliably when executed that it cannot put the system into an unsafe mode. An example here would be that the software gets hung up in a loop and the safety functionality is never invoked.
- **Category 8.** Suggests safe but not secure software behavior. This is quite realistic for a safety-critical system with no security concerns. Note that the interesting aspect of this category is how safety and security are defined. Many people use these terms interchangeably, which is incorrect.
- **Category 11.** Suggests that the software behaves reliably and has good availability, but lacks adequate security precautions. Many systems suffer from this problem.
- **Category 12.** Suggests that the software behaves reliably, is extremely slow, but has adequate security. It makes one wonder if the system is so slow that it is effectively unusable, and thus secure, since it would take too long to break in.
- **Category 13.** Suggests high levels of security and high levels of performance. In certain situations that is plausible, however typically security kills performance and vice versa.
- **Category 14.** This is the easiest combination to achieve. Anyone can build a useless system.

The main point here is that the aforementioned high-level attributes (1) produce accurate and reliable output, (2) produce the needed output in a timely manner, and (3) produce the output in a secure and private manner are actually composed of the lower-level *ilities*. Another important point not to overlook is the fact that some of the *ilities* are not compatible with one another. An example of this can easily be found using fault tolerance and testability. A final important point is that some combinations of the *ilities* are simply counterintuitive, such as a system that is safe but unreliable.

One last thing to note: It is vital to get solid definitions for the *ilities* and to know which ones are quantifiable. For example, reliability and performance are quantifiable; security and safety are not. This makes it far easier to make statements such as *we have very high reliability but an unknown level of security.*

The Shall Nots

There is yet another layer in the notion of fit for use that deals with *negative* functional requirements. Think of a negative requirement as "the software shall *not* do X," as opposed to a functional requirement stating that "the software *shall* do X."

"For certain types of systems, particularly safety-critical systems, enumerating negative requirements is a necessity. And for software requiring security capabilities, security rules and policies are its equivalent to negative requirements."

Negative requirements are far more difficult to elicit than regular requirements. Why? Because humans are not programmed to anticipate and enumerate all of the bad circumstances that can pop up and that we need protection against; we are instead programmed to think about the *good* things we want the software to do.

For certain types of systems, particularly safety-critical systems, enumerating negative requirements is a necessity. And for software requiring security capabilities, security rules and policies are its equivalent to negative requirements. For example, a negative security requirement could be that the software shall never open access to a particular channel unless it can be guaranteed that the information passing through the channel is moving between trusted entities. The difficulty in defining *shall not's* for security is that we cannot imagine all of the different forms of malicious attacks that are being invented on-the-fly and if we cannot imagine those attacks, we likely will not prevent them.

Before leaving the topic of negative functional requirements, it is worth mentioning an interesting relationship between them and the environment. So far, we have only mentioned traditional inputs and phantom users as players in the environment. Traditional inputs are those that the software expects to receive during operation. But there are two other types of inputs worth mentioning: *malicious illegal* and *non-malicious illegal*. A malicious illegal input is one that someone deliberately feeds into the software to attack a system, and a non-malicious illegal input is simply an input that the system designers do not want the software to accept but has no malicious intent. In both cases, filtering on either type of input can be useful to ensure that certain inputs do not become a part of the environment and in doing so ensure that negative functional requirements are enforced.

Time

The next layer in our quest for software quality is *time*. Software has fixed longevity; it can be expanded, as we learned from Y2K, but not indefinitely.

One of the easiest ways to explain why time fits here is to look at the situation where a software package operates correctly on Monday but does not operate correctly on Tuesday. Further, the software package was not modified between these days. (This is the classic problem that quickly carves down the number of freshman computer science majors.) Why has this problem occurred?

It all goes back to the importance of environment in the understanding of software quality. Earlier we defined the environment as inputs with probabilities of selection, hardware configurations, access to memory, operating systems, attached databases, and whether other background processes were over-indulging in resources, etc.

But what is not mentioned was *calendar time*. Environment is also a function of time. As time moves forward, other pieces of the environment change. And so while all effort and expense can be levied toward what we perceive is evidence supporting the claim that we have good enough software, we need to recognize that even if we do have good enough software, it may be only for a short window of time. Thus, software quality is *time-dependent*, a bitter pill to swallow.

Cost

We cannot end this article without mentioning *cost*. The costs associated with software quality are exasperated by the un-family like behaviors of various *ilities*. Not only are there technical trade-offs discovered when trying to increase the degree to which one *ilitiy* exists only to find that another is automatically decreased, but there is the financial trade-off quagmire concerning how to allocate financial resources between distinct *ilities*. If you overspend on one, there may not be enough funds for another. And, as if the technical considerations are not hard enough when trying to define software quality, the financial considerations come aboard, making the problem worse.

Conclusion

In this article, a set of layers for what software quality means has been unpeeled. I have argued that a more useful perspective for what software quality represents starts from the notion of the software being *fit for purpose*, which requires:

1. Understanding the relationship between the *functional requirements* and the *environment*.
2. Understanding the three high-level attributes of software quality: the software: (a) produces *accurate and reliable* output, (b) produces the needed output in a *timely* manner, and (c) produces the output in a *secure and private* manner.
3. Understanding that the i*lities* afford the potential to have varying degrees of the high-level attributes.
4. Understanding that the *shall-not* functional requirements are often of equal importance to the functional requirements.
5. Understanding that there is a *temporal* component to software quality; software quality is not static or stagnant,
6. Understanding that the *ilities* offer technical incompatibilities as well as financial incompatibilities.
7. Understanding that the environment contains many more parameters such as the phantom users than is typically considered.

Thus, software quality, when viewed with these different considerations, becomes a far more interesting topic, and one that will continue to perplex us for decades to come.◆

References

1. IEEE. Standard Glossary of Software Engineering Terminology IEEE. American National Standards Institute/IEEE Standard 729-1983: 1983.
2. Marciniak, J., ed. Encyclopedia of Software Engineering. Second Edition. Wiley Inter-Science: 2002.
3. Wieczorek, Martin, and Dirk Meyerhoff, editors. Software Quality: State of the Art in Management, Testing, and Tools. Springer. New York: 2001.
4. Gao, J.Z., H.S. Jacob Tsao, and Y. Wu. Testing and Quality Assurance for Component-Based Software. Artech House. Norwood, MA: 2003.
5. Whittacker, J., and J. Voas "50 Years of Software: Key Principles for Quality." IEEE IT Professional. 4(6): 28-35, Nov. 2002.
6. Musa, John D., Anthony Iannino, and Kazuhiiro Okumoto. Software Reliability: Measurement, Prediction, Application. McGraw-Hill, NY, 1987.
7. Voas, J. "Software Certification Laboratories?" CROSSTALK Apr. 1998.
8. Voas, J. "Can Clean Pipes Produce Dirty Water?" IEEE Software July 1997.
9. Voas, J. "Software's Secret Sauce: The 'Ilities." IEEE Software Nov. 2004.

About the Author

Jeffrey Voas, Ph.D., is currently director of systems assurance at Science Applications International Corporation (SAIC). He was the president of the IEEE Reliability Society from 2003-2005, and currently serves on the Board of Governors of the IEEE Computer Society. Voas has published numerous articles over the past 20 years, and is best known for his work in software testing, reliability, and metrics. He has a doctorate in computer science from the College of William & Mary.

SAIC
200 12th ST South
STE 1500
Arlington, VA 22202
Phone: (703) 414-3842
Fax: (703) 414-8250
E-mail: j.voas@ieee.org

Forecasting the Future

Welcome to the June issue of CROSSTALK – and I hope you didn't miss the Systems and Software Technology Conference (SSTC) conference in Las Vegas about a month ago. It was great! First of all, the location was superb – conference facilities, hotel, location. Come on – Barry Manilow *and* the Star Trek Experience? Geek heaven. The weather was fantastic (although a bit warm during the day), and having the monorail to travel to/from the Strip was convenient.

The exhibits were also good (as usual). There seemed to be quite a few of the *process-oriented* vendors this year – a good thing, if you ask me. And, let's face it, the giveaways and gadgets were spectacular. The food was also great. Personally, I think that the speakers were better than ever this year. It was good seeing a lot of old friends, and making new ones. The only drawback was that with all of the distractions, far too many of us stayed up late, and made the 2008 SSTC conference a conference to remember. What more could you ask for?

Except that I am writing this column in March, and the SSTC conference is still a month in the future. However, I am totally convinced that almost everything I wrote above *will* be true, and that after the conference ends, I *will* be able to argue that I was *very* successful in predicting the future. If only software were so easy.

The theme of this issue is Software Quality. I have a very unique definition of quality. In my mind, quality was defined by Simon and Garfunkel back in 1970 on their "Bridge Over Troubled Water" album (arguably the best piece of music *ever* released). There was a song entitled "Keep the Customer Satisfied." Awesome lyrics. And that is the key – keeping the customer satisfied.

So, how do I achieve this dubious thing called quality which is hard to measure directly? Well, I am sure that this issue contains great articles about quality (since I'm writing in the future, I don't even know the article lineup yet!).

In my mind, quality needs to include customer satisfaction. Is the software going to be used in a passenger aircraft? Well then, as a potential customer, I am pretty darn hard to satisfy. The other day, I was flying "across the little pond" (returning from London across the Atlantic) and the in-seat entertainment system I was using crashed – I actually got a *core dump* error message and saw Linux rebooting. The lady sitting next to me had it happen to her, and her comment was, "I sure hope that the software that runs the aircraft works better." Well, having worked with several aircraft software developers, I can assure you that it *does* run much better. Is the software going to be used to simply rip a few of my old CDs to MP3s so that I can load them onto my latest gadget? Then I am willing to have it occasionally fail. I have a feeling that the latest SuperX MP3 Ripper program I downloaded free off the Web cost a *lot* less per line of code to develop than the software that will power the Joint Strike Fighter.

Which brings me back to forecasting the future. Forecasting the future is not an exact science. A friend of mine who is a meteorologist says that an 85 percent to 90 percent success rate in intermediate range predictions (one to three days out) is great. Two weeks out? More general predictions (i.e., *warming trend*) give the forecaster some leeway. One day out? A high of 73 with afternoon showers, ending by 9 p.m. tonight.

The secret to quality is the same: if you *really* think you can set a schedule (such as 28 lines of code per programmer per day) that will allow your developers to achieve a quality target ("No more than two errors found per 500 function points during integration testing") that is also reasonably accurate one year out, well ... how did you enjoy SSTC 2009? Even with *lots* of historic data for similar projects, each development effort is different. Weather forecasters have access to about a hundred years of hurricane data, but still cannot tell me one month out when and where (or even if) a hurricane is going to hit the United States.

Quality is fragile. It is hard to achieve and, once lost, it seems to be gone forever. You can't test quality back in – those who have tried know better. You have to plan aggressively for quality and you have to have a good process for it (I wasn't kidding earlier – the more process-oriented vendors at SSTC, the better it is for Department of Defense software in general).

Forecast the future as best you can. Revise your forecasts (and timelines) as you get closer to your goals. A good weather person has no problem saying, "Well, last week we said generally clear, but prepare for a heavy rain tomorrow." A good program manager might just have to say, "We had hoped to be in integration testing this week, but we need another month to complete inspections and peer reviews." Nobody *wants* to hear that their vacation and beach plans are going to be washed away, but it happens. Nobody *wants* to hear that we are having problems with code quality, but if you have a good forecast and update it as needed, maybe you won't.

Hope you enjoyed SSTC 2008 – and see you at SSTC 2009. Trust me – it was great, also!

—David A. Cook
The AEgis Technologies Group, Inc.
dcook@aegistg.com

Can You BACKTALK?

Here is your chance to make your point, even if it is a bit tongue-in-cheek, without your boss censoring your writing. In addition to accepting articles that relate to software engineering for publication in CROSSTALK, we also accept articles for the BACKTALK column. BACKTALK articles should provide a concise, clever, humorous, and insightful perspective on the software engineering profession or industry or a portion of it. Your BACKTALK article should be entertaining and clever or original in concept, design, or delivery. The length should not exceed 750 words.

For a complete author's packet detailing how to submit your BACKTALK article, visit our Web site at <www.stsc.hill.af.mil>.

Homeland Security

Software Assurance Program

Software is essential to enabling the nation's critical infrastructure. To ensure the integrity of that infrastructure, the software that controls and operates it must be reliable and secure.

Visit https://www.us-cert.gov/SwA to learn more about the software assurance program and how you can become more involved.

Security must be "built in" and supported throughout the lifecycle.

Visit https://BuildSecurityIn.us-cert.gov to learn more about the practices for developing and delivering software to provide the requisite assurance.

Sign up to become a free subscriber and receive notices of updates.

BuildSecurityIn
Build Security In Home
Software Assurance
Community Resources and Information Clearinghouse
Sponsored by DHS National Cyber Security Division
HOME
PEOPLE
PROCESS
TECHNOLOGY
ACQUISITION
WORKING GROUPS
US-CERT Software Assurance
Workforce Education & Training
Processes & Practices
Technology, Tools & Product Eval.
Acquisition & Outsourcing
Measurement
Business Case
Malware
Join a Working Group
Build Security In

Software Assurance Working Groups

The Department of Homeland Security National Cyber Security Division has established stakeholder working groups to support each of its Software Assurance Program's focus areas. The goal of the working groups is to continue to support Software Assurance initiatives by bringing together members of government, industry, and academia with vested interests in software security to discuss and promulgate applicable practices and methodologies.

Software Assurance Focus Area and Working Group Matrix

https://www.us-cert.gov/SwA

The Department of Homeland Security provides the public-private framework for shifting the paradigm from "patch management" to "software assurance."

CrossTalk / 517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737

CrossTalk is co-sponsored by the following organizations:

NETWORKS AND INFORMATION INTEGRATION
ASSISTANT SECRETARY OF DEFENSE
DOD CIO

DEPARTMENT OF DEFENSE
UNITED STATES OF AMERICA

NAVAIR

76th SOFTWARE MAINTENANCE GROUP

309th SOFTWARE MAINTENANCE GROUP

U.S. DEPARTMENT OF HOMELAND SECURITY
Homeland Security